Faust: An Integrated Environment for Parallel Programming

Vincent A. Guarna, Jr., ***Dennis Gannon***, ***David Jablonowski***, ***Allen D. Malony***, and ***Yogesh Gaur***, *University of Illinois at Urbana-Champaign*

Designed for the development of large, scientific applications, Faust includes several new tools and integrates existing tools. Some components will be ready for public distribution this year.

Today, many environments are being constructed to coordinate the disjoint activities of editing, debugging, and tuning complex applications designed to run on parallel architectures. Faust is a workstation-based programming environment for scientific applications being developed at the Center for Supercomputing Research and Development at the University of Illinois.

(Although Faust was named with no underlying acronym or rationale, it has occurred to us that completing the project may require a deal with the devil.)

Faust is intended to provide a tool set for programming parallel machines. We have three major goals:

- To design and implement a set of new tools specifically designed to help develop efficient parallel programs. This includes interactive compilation and optimization tools and facilities for debugging and analyzing performance in a parallel environment.
- To integrate these new tools with existing tools such as system text editors and compilers without modification. To be effective, a computing environment must offer an integrated set of functions and a uniform user interface.
- To ensure portability. Although Faust's basic platform is a bitmapped workstation running Unix, we expect it to run on a variety of hardware. To accomplish this, we have layered all user-interface libraries on top of the X Window System and we have designed all file operations to work on a single file-namespace system, such as NFS from Sun Microsystems.

Architecture. Figure 1 shows Faust's organization. At the user level are application-development utilities. Included at this level are traditional Unix development tools such as system text editors and compilers and Faust's parallel-programming tools such as a performance-evalua-

0740-7459/89/0700/0020/$01.00 © 1989 IEEE

tion facility and interactive compilation tools.

As Figure 1 shows, these tools access the file system through the Project Manager, a hierarchical database manager that Faust uses to associate related objects in way that conceals the network. The Project Manager also provides a locking mechanism that lets you share project components inside and outside Faust.

At the lower right in the diagram is Faust's user interface, shown as a layer of building blocks available to the Faust tools. The building blocks comprise interface utilities that do basic I/O operations with X Windows. This layer also maintains the hierarchical program abstraction, which lets you view programs in varying levels of detail — from a low-level textual view of source code to a high-level graphical view of function and task relationships.

Project database

In Faust, all applications work is done in the context of projects. The project is the unifying theme in Faust, serving as the focal point for all tool interactions. A project roughly corresponds to an executable program. Faust achieves functional integration through operations on common data sets maintained in each project.

Project Manager. The Project Manager organizes and manipulates project components. These components, called objects, typically represent Unix files. A simple project might consist of a single executable program and its associated source, object, and include files. A complex project might include many executable programs, libraries, and data repositories from tools like the performance analyzer.

Projects and objects are identified in an object name space that is independent of the file name space. This lets you assign object names that relate to the project instead of forcing you to use file names with directory prefixes that have been established by a system administrator. An independent object name space also means that the location of a physical file, and thus its network path, can change while the associated object path remains constant.

This naming scheme is important because Faust is designed to be a multiuser, distributed, heterogeneous environment, where distributed project creation and file migration are likely. By keeping object names constant, Faust insulates users and tools from the details of absolute file paths and machine names.

In Faust, the project is the unifiying theme, serving as a focal point for all tool interactions.

The Project Manager organizes a project's objects by building a directed graph, where nodes are objects and arcs are named relationships between objects. Applications and users control a relationship's name and the objects associated with it. For example, an object representing a library would be related to all its constituent object-file objects via the relationship named obj.

In the spirit of the Unix Make utility, you can define a relationship to be a time-based dependency; this supports object consistency. An object is consistent if its last-modified time is more recent than the last-modified times of all of the files on which it depends. If an object is found to be inconsistent, it can be made consistent by executing a script of commands associated with it.

Such a build script can contain commands for any node in the heterogeneous system; Project Manager servers wait on each node to execute the commands for that node. By creating scripts that communicate with these special demons on

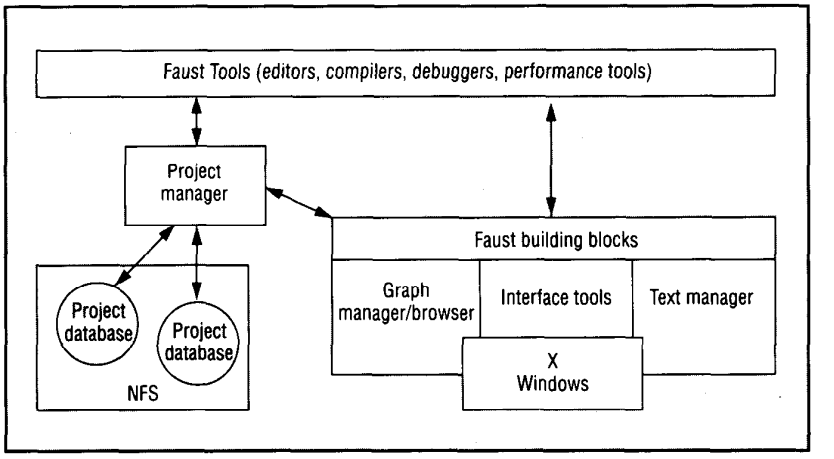

Figure 1. Faust organization.

Comparing Faust

The labels attached to programming environments are so ill-defined as to be almost interchangeable: scientific-programming environments, software-engineering environments, software-development environments, and so on. Indeed, a survey by Fedchak[1] found that the development of these environments is driven by specific needs and goals.

Therefore, we compare Faust with four development efforts driven by the same goals.

ParaScope. ParaScope, an extension of the R^n environment, is being developed at Rice University.[2] ParaScope focuses on restructuring sequential Fortran to parallel form. It supports both automatic and manual, interactive restructuring.

ParaScope has integrated restructuring editors, compilers, and a parallel debugger. While Faust is also designed to help restructure sequential code, it is more flexible than ParaScope in that it lets you integrate arbitrary tools into its environment.

Sigma is more closely related to ParaScope than to two other restructuring tools, Ptran[3] and Parafrase II,[4] both of which work on a database of program dependency and interprocedural information.

AISPE. The framework of the Advanced Industrial Software Production Environment is more general than Faust's.[5] AISPE is a software-production environment that attempts to support the entire life cycle. AISPE acts as external shell around the operating system to filter user commands and actions. This shell lets developers integrate commercially available tools.

AISPE's object handler manages objects that constitute projects. In AISPE, objects undergo state transitions, using high-level petri nets as control structures. Although AISPE's integration efforts are more ambitious than Faust's, it addresses neither parallel programming nor distributed processing in a heterogeneous system.

MicroScope. Hewlett-Packard's MicroScope project uses a knowledge base for project management.[6] The knowledge base consists of frames and inference rules. A frame's data may include procedural scripts or methods for computing objects in the knowledge base.

MicroScope is concerned with effectively conveying program structure through views. A view may show the relationships between a program's modules or a static analysis of procedure relationships. Faust also uses abstractions to give the user a graphical view of file and subroutine relationships.

SIB. The Software Information Base was developed by General Telephone and Electric.[7] SIB is an elaborate data model, or object base, that stores a project's data over the entire life cycle. SIB seeks to improve integration: Its objects can have types, classes, supertypes and subtypes, and superclasses and subclasses. In the spirit of formal database theory, SIB defines its model's internal, conceptual, and external layers.

Because of its generality and complexity, SIB has had performance and user-interface problems in its initial prototypes.

Both MicroScope and SIB share some of our project-management goals, with SIB's goals being much more ambitious. However, how these goals are achieved are very different in all three environments. MicroScope's and SIB's primary goals are limited to project management. Faust uses project management to provide high-level integration for development tools, but its overall goals are much broader.

References

1. E. Fedchak, "An Introduction to Software Engineering Environments," *Proc. Computer Software and Applications Conf.*, CS Press, Los Alamitos, Calif., pp. 456-463, 1986.
2. D. Callahan et al., "ParaScope: A Parallel-Programming Environment," Tech. Report Comp TR88-77, Rice University, Houston, 1988.
3. F. Allen et al., "An Overview of the Ptran Analysis System for Multiprocessing," *Supercomputing*, E.N. Houstis, T.N. Papatheodorou, and C.D. Polychronopoulos, eds., Springer Verlag, Berlin, pp. 194-211.
4. C. Polychronopoulos et al., "Parafrase II: A New-Generation Parallelizing Compiler," *Proc. Int'l Conf. Parallel Processing*, CS Press, Los Alamitos, Calif., to appear, 1989.
5. G. Bruno, P. Spiller, and I. Tota, "AISPE: An Advanced, Industrial Software-Production Environment," *Proc. Computer Software and Applications Conf.*, CS Press, Los Alamitos, Calif., pp. 94-99, 1986.
6. J. Ambras and V. O'Day, "MicroScope: A Knowledge-Based Programming Environment," *IEEE Software*, May 1988, pp. 50-58.
7. J.H. Kuo and H.-C. Tu, "Prototyping a Software Information Base for Software-Engineering Environments," *Proc. Computer Software and Applications Conf.*, CS Press, Los Alamitos, Calif., pp. 38-44, 1987.

other machines, the Project Manager implements a remote compilation and execution facility.

Because Faust is a multiuser, distributed environment, the Project Manager includes a central server to administer locks. All Faust tools access the project database by first requesting arbitration and file-location services from the Project Manager, then using conventional Unix file I/O.

Named relationships among objects make it possible for the Project Manager to answer queries from the Faust tools. In the obj relationship, for example, an executable program's object files are contained in the set of objects that are destination objects in the obj relationship. These objects can all be reached from the object that represents the executable. Such queries are most often made by the Faust performance tools, which must modify a program's object files to generate data at runtime.

Database components. The Project Manager maintains eight types of files for every Faust application:

- Executable: The ultimate target object.
- Source: The original program text written in Fortran or C.
- Object: Intermediate files generated by system compilers when they produce an executable program.
- Assembler: Assembly-language versions of the source produced by the system compilers. The Project Manager creates these for reference by the performance-prediction tools.
- Dependency: Symbol-table and data-dependency information collected by Faust compilers for reference by the restructuring environment described later.
- Program graph: A static call graph used by Faust's graphical browser.
- Execution trace: Collected at runtime as a result of monitoring by performance-evaluation tools. These trace files are referenced by performance-analysis and visualization tools.
- Annotation: Detailed information about modifications applied to applications on behalf of Faust tools. For example, every execution trace done by performance-analysis tools has an annotation file that contains a detailed description of

the performance data collected and the reason for collecting it.

Graphical Make file. One development tool that you can easily construct with Faust's building blocks is a graphical version of the Unix Make utility.

Faust's graphical Make-file editor lets you create a directed graph to show program dependencies, as shown in Figure 2. At the root of the graph, or tree, is the executable object. The next level contains all the object files needed to generate the executable object. Each object file is itself the root of a subtree that contains all the files needed to generate it.

The graphical Make-file editor highlights those executable files that are out of date or inconsistent by drawing a box around their node, as Figure 2 shows. This reminds you which recompilations you must perform. You can specify that a certain subtree be recompiled or you can let the system perform all necessary build operations.

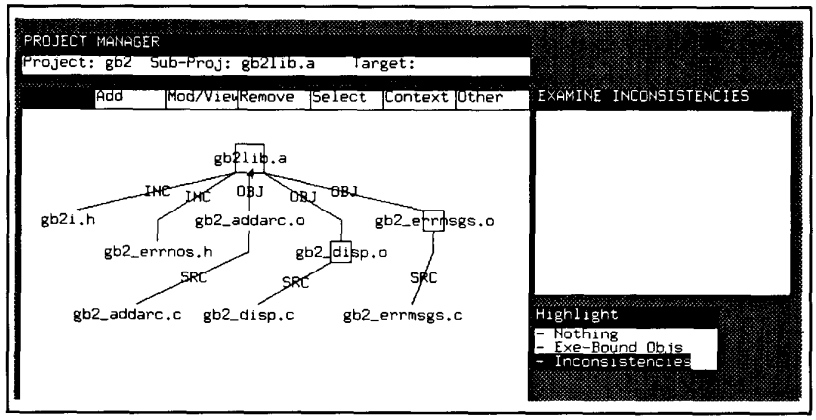

Figure 2. Using the graphical Make-file editor to create a directed graph of program dependencies. Nodes that are out of date or inconsistent are boxed.

User-level tools

Sigma[1] is a Faust tool designed to help users of parallel supercomputers retarget and optimize application code. Sigma helps you either fine-tune parallel code that has been automatically generated or optimize a new parallel algorithm's design.

At its lowest level, Sigma is a mouse-based, multiwindow text editor with a shell interface that can be used the way most programmers use Emacs. (In fact, we are developing an Emacs front end.) Sigma's power, however, lies in its interface to the Faust program database.

An application's project database contains

- a complete data-dependency analysis (both inner and interprocedural) of the application's source files,
- a control-flow graph with enough information to regenerate the original source file (including comments), and
- a summary and analysis of the object code generated by the compiler.

The database can support either Fortran with Alliant's vector and parallel extensions, Cedar Fortran (a Fortran 8X extension designed to exploit the Cedar multiprocessor), and C. The database also supports a parallel, object-oriented C extension similar to C++ and Cedar Parallel C. The only target machine that now supports object-code analysis is the Alliant FX/8, but we are working on an analyzer for the BBN Butterfly and Ardent Titan.

After the Project Manager builds an application, it uses special parsers to generate a project database, which you query with the Sigma editor. For example, when an application's source file is displayed, you can select (with a mouse) an item such as a variable or function name and make queries and issue commands like:

- Where was this variable initialized or last modified?
- Which routines modify or use this variable?
- What side effects does a call to this procedure or function generate, and which segments of array parameters are used or modified?
- Can this loop be parallelized or vectorized? If not, which variables prohibit concurrency?
- If this variable is a pointer to a C structure (or object in a C++ class), what are the fields (operators) in that structure (object)?
- Generate an estimate of the caches' hit ratios for array objects in this code segment.
- Generate an estimate of code efficiency (measured in floating-point operations per second) for this code segment.
- Draw this function's static call graph.
- Draw this code segment's data-dependency graph.

With this kind of access to an application's semantics, you can work with the system to restructure the code for a target architecture.

In addition to semantic data, the graphical representation of the code's internal form lets you guide the system in transforming the form. In this mode, you select a program segment you want to modify. Faust then presents you with a menu of predefined program transformations, including loop vectorization, parallelization, interchanging, blocking, distribution, and some machine-specific transformations. We are adding other menu options, including subroutine expansion and encapsulation and variable localization. If you try to transform the program in a way that violates its original semantics, you are warned that the transformation will change the program's meaning.

Sample scenario. Sigma is designed to help users port large applications to parallel computers.

In any porting effort, the first step is to use any automatic restructuring tools that are available. For large programs, these tools will sometimes provide the needed performance without significant programmer effort. Unfortunately, these tools often fail to extract medium- and coarse-grained parallelism from the program's higher levels. Yet this is often the type of parallelism that is best supported by multiple-instruction, multiple-data machines like Cedar.

Therefore, after the code has been compiled and the performance information loaded into the database, the programmer may want to improve on the parallelization. The next step is to begin restructuring the program by transforming code segments to express more parallelism.

As a simple example, Figures 3 and 4 show a Sigma session in which the programmer is investigating a simple matrix times a vector subroutine. In Figure 3, the

Figure 3. The Sigma restructuring tool. The programmer has highlighted the Do *i* loop for object-code analysis. The Edit Transcript window shows the object-code summary.

programmer has selected the Do *i* loop for object-code analysis. The Edit Transcript window shows a summary of the restructuring that was done automatically by the Alliant compiler and an algebraic expression representing the cycle count for the loop in terms of the number of processors (denoted by the symbol #p) and the loop bounds. The summary lists the concurrent and vector looping structure, the number of scalar instructions (denoted by S:), the number of vector instructions (denoted by V:), and the total cycle count for the corresponding basic block.

If he decides that the compiler did a poor job, the programmer can restructure the segment to produce the program in Figure 4. The first transformation interchanged the Do *i* and Do *j* loops. The second transformation blocked the Do *i* loop by a factor of 32, generating a Do i_T loop. This caused the innermost loop to have a length ideal for the vector registers in the Alliant FX/8. In doing this, the compiler generated two temporaries, zt0 and zt1, which are part of the index computations. The Do i_T loop was then carried to the outside by another transformation and the innermost loop was vectorized. All of these actions are carried out by by selecting the loop to be modified and then selecting the the appropriate action from the Transform menu.

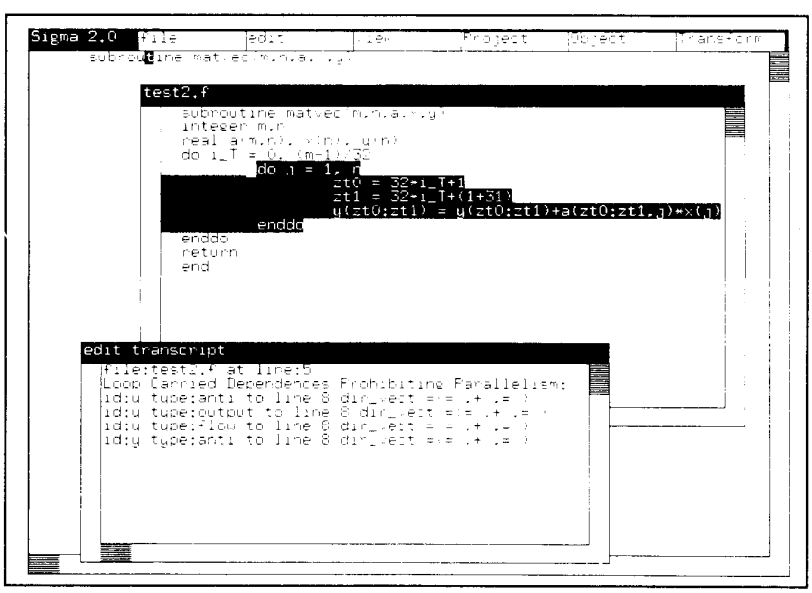

Figure 4. Restructuring the segment from Figure 3 to improve parallelization.

To illustrate Sigma's ability to check the semantics of a transformation, in Figure 4 the programmer has requested that the inner loop be parallelized. The system refused to complete the transformation for the reasons shown in the Edit Transcript window.

Performance evaluation. Faust includes a dynamic call-graph tool.[2] This tool shows an animated view of a program's execution to help you understand its operation, similar to techniques described by Meyers[3] and Brown.[4] The dynamic call-graph tool displays the dynamic execution of a parallel program in the context of the original subroutine interconnection graph that Faust generates automatically.

Figure 5 shows 10 windows from Faust's graph browser, each showing some part of a program's execution. The target system for this display is the Alliant FX/8 with eight processors. The first eight windows show the state of execution on each processor. The Sequential window shows the state of the program's call graph when it transitioned from a sequential-execution mode to a parallel-execution mode. The Global window shows the union of the views.

Impact. Faust's Integrated, Multiprocessor Performance-Analysis and -Characterization tool set integrates its tools that collect performance data with tools that analyze and display performance results. Lately, we have concentrated on developing Impact's event-display tool. This tool traces multitasking events and displays them in a time line.

Figure 6 shows a screen of the Impact event-display tool. In the main control window (upper left), the user can select the trace directory, which stores the traced files; the event-definition file, which describes the event types found in the trace; and the event-trace file, which contains a time-sequenced list of all generated events. The Project Manager chooses some default directories and paths, but you can override these defaults.

After you specify the files, Impact first

reads the event-definition file and then the trace-data file, using the event descriptions to interpret the trace input. After it reads all the trace data, Impact generates the global-trace statistics, shown in the bottom half of the screen. Impact also creates an internal trace-file index to support rapid event searches and it maintains an event cache to reduce disk transfers.

A task group is responsible for displaying the events associated with a task. A task group defines a time window for the trace. Events for tasks that have been assigned to a task group by the user will be displayed in the task display only if the time at which they occurred is within the task group's time window.

In the display, the task group is shown at the bottom and is represented as a time line. The controls to change the time window are the Start, <<<<, Zoom In, Zoom

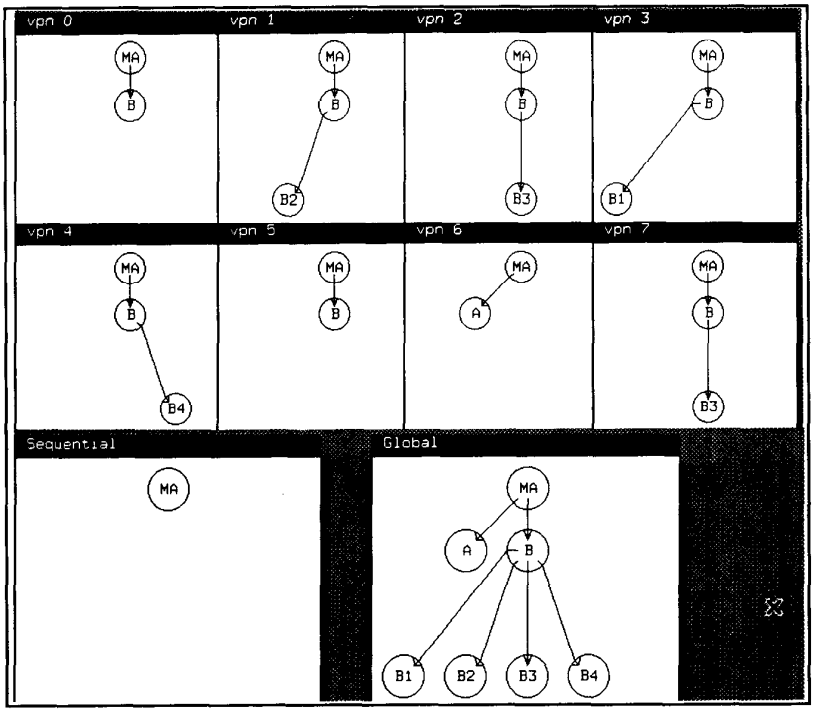

Figure 5. The dynamic call-graph browser. The top eight windows show the execution on each processor. The Sequential window shows the state of the call graph when it was transitioned and the Global window shows the union of the views.

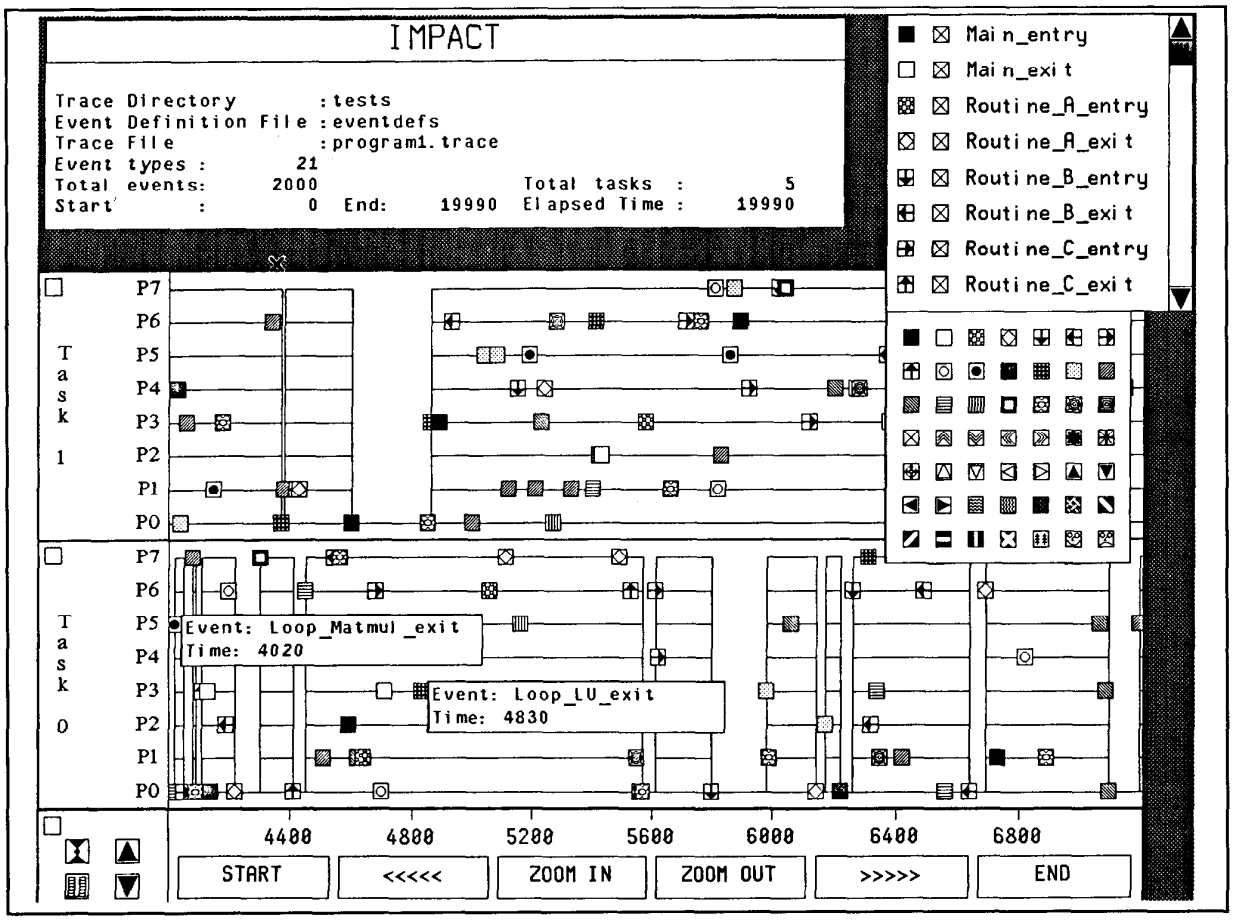

Figure 6. The Impact event-display tool.

Out, >>>>, and End buttons.

Faust provides icons for viewing the entire task group and its associated displays, scrolling between viewable tasks in the group, scrolling between viewable tasks, and attaching new tasks to the group. Finally, Faust lets you view multiple task groups in different time windows simultaneously.

Each task display shows events for each active processor. (Impact will be used on the Cedar machine, where tasks execute on Alliant FX/8 clusters and may use all eight processors on each cluster.) Impact draws a horizontal line for each active processor. When the execution is sequential, Impact draws a horizontal line for processor 0 only.

In Impact, every event specified in the event-definition file has a unique icon. Events are placed in the task display at the point given by the processor ID where it occurred and the time when it occurred. Depending on the time-window's resolution, event icons may overlap. As Figure 6 shows, you can request details about an event by clicking on its icon (shown for two events in Task 0). This is useful when you forget what an icon represents or when you want more data about the event.

In the event-control window in the upper right, users can change which events will display and how they display. This window lists every event from the event-definition file, its event-icon assignment, and the state of its visibility. You click on the event icon to get a palette (shown just below the event-control window) of icons from which you can select a new icon. You click on the visibility box to toggle the event's visibility in the display.

In the event-control window you can also categorize events and abstract them into groups, such as subroutine entry, which are treated as a single logical event in the display. For example, making the icon for all events a black box gives you a sense of the number and density of events in a time window without the confusing clutter of all the event icons. Similarly, making all events invisible will show you the processor activity lines only, from which you can readily identify sequential and concurrent execution transitions.

The Impact event-display tool also lets you search for events by task group, set marks at different points in time, and dump task displays to files. We plan to develop displays that will use the event data to view task-level activity and will show results from more extensive event-trace analysis. We also plan to implement a single thread task display.

We have used the Impact event-display tool in several performance studies, such as the design of a parallel circuit simulation code. Our initial reaction is that observing a program's event sequence is valuable in identifying performance-limiting behavior. However, we need to better integrate Impact with other Faust tools so a user can access all the information about a program in a common framework.

Faust is far from finished. We are constructing other tools, including a debugger, a roundoff-error analyzer, and an on-line librarian. We expect to continue adding and revising tools for a long time.

Several tools have been deployed internally at the Center for Supercomputer Research and Development as alpha releases. These include the graphical Make file, Project Manager, Sigma editor, and the dynamic call graph and Impact tools. We plan to make a public distribution of some components later this year.

Reponse to the Sigma editor has been good, although we can't implement it fully until we complete the integration. We are particularly interest in studying which benefits users derive from each Sigma feature. We want to make a serious analysis of each feature's contribution because each one is computationally expensive. Keeping the features to a minimum will help keep Faust interactive.

Our initial experience with the other tools has been good. We have ported the Faust building blocks — developed on Apollo workstations — to Sun workstations and IBM PC RTs without problems. We think X Windows has worked well as a user-interface platform. ❖

Acknowledgments

Several people are responsible for the Faust project. Bruce Shei, Daya Attapatu, and Jenq Kuen Lee designed and implemented Faust's database tools at Indiana University. We also thank James Krause, David Kuck, Duncan Lawrie, Paul Lewis, Joseph Pickert, and Dan Reed for their efforts and continuing support.

Faust is supported in part by US Air Force Scientific Research Office grant AFOSR-F49620-86-C-0136, National Science Foundation grant MIP-8410110, US Energy Dept. grant DE-FG02-85ER25001, and IBM.

References

1. D. Gannon et al., "A Software Tool for Building Supercomputing Applications," *Proc. Parallel Computations and Their Impact on Mechanics*, ASME, New York, 1987, pp. 81-92.
2. K.A. Gallivan et al., "Performance Analysis on the Cedar System," in *Performance Evaluation of Supercomputers*, J.L. Martin, ed., North-Holland, Amsterdam, 1987.
3. B.A. Meyers, "Incense: A System for Displaying Data Structures," *Computer Graphics*, July 1983, pp. 115-125.
4. M.H. Brown, "Techniques for Algorithm Animation," *IEEE Software*, Jan. 1985, pp. 28-39.

Vincent A. Guarna, Jr., is a senior software engineer and manager of the Faust project at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. His research interest include programming environments, parallel-programming tools, and restructuring compilers for C.

Guarna is a doctoral candidate in computer science at the University of Illinois. He received a BS in computer science from Roosevelt University, Chicago, and an MS in computer science from the University of Illinois.

Dennis Gannon is an associate professor in computer science at Indiana University and part-time researcher at the Center for Supercomputing Research and Development. His research interests are algorithm design, performance analysis, tools, and graphics for parallel processing.

Gannon received a PhD in mathematics at the University of California at Davis and a PhD in computer science at the University of Illinois.

David J. Jablonowski is a senior software engineer at the Center for Supercomputing Research and Development, where he specializes in integrated environments.

Jablonowski received a BS in computer science from the University of Wisconsin at Eau Claire and an MS in computer science from Boston University. He is a master's candidate in applied mathematics at the University of Illinois.

Allen D. Malony is a senior software engineer at the Center for Supercomputing Research and Development, where he is manager of performance-evaluation of the Cedar multiprocessor. His research interests are parallel computation, multiprocessor architectures, and performance evaluation.

Malony is a doctoral candidate in computer science at the University of Illinois. He received a BS and an MS in computer science from the University of California at Los Angeles.

Yogesh Gaur is a doctoral candidate at the University of Illinois. His research interests are parallel computing, programming environments, and restructuring compilers.

Gaur received a BTech degree in electrical engineering from the Indian Institute of Technology, Delhi, and an MS in computer science from the State University of New York at Buffalo.

Address questions about this article to Guarna at the Center for Supercomputing Research and Development, 305 Talbot Laboratory, 104 S. Wright St., Urbana, IL 61801.