Program Analysis Environments for Parallel Language Systems: The $\mathcal{T}$ Environment [*]

Darryl Brown, Steven Hackstadt, Allen Malony, Bernd Mohr[†]

Abstract

In this paper, we discuss $\mathcal{T}$ (TAU, **T**uning and **A**nalysis **U**tilities), the first prototype of an integrated and portable program analysis environment for pC++, a parallel object-oriented language system. $\mathcal{T}$ is unique in that it was developed specifically for pC++ and relies heavily on pC++'s compiler and transformation tools (specifically, the Sage++ toolkit) for its implementation. This tight integration allows $\mathcal{T}$ to achieve a combination of portability, functionality, and usability not commonly found in high-level language environments. The paper describes the design and functionality of $\mathcal{T}$, using a new tool for breakpoint-based program analysis as an example of $\mathcal{T}$'s capabilities.

1 Introduction

The trend towards using high-level parallel language systems to program scalable parallel computers must be accompanied by advances in the tools and environments for program analysis and tuning. The language system concerns are achieving programmability through parallel programming abstractions, portability through retargetable runtime systems, and scalability and performance through restructuring and optimizing compiler tools. These concerns lead to a new set of requirements for program analysis support: analysis based on language/program semantics, portable measurement and analysis infrastructure, and compiler-integrated analysis. These requirements are difficult to address using general-purpose parallel tools because of the tight logical coupling that is necessary between the language model and program analysis functionality. Moreover, for evaluating the usability and programmer productivity issues associated with parallel program analysis tools it is important to have an application- or language-specific context.

$\mathcal{T}$ (TAU, **T**uning and **A**nalysis **U**tilities) is a parallel program analysis environment that we have designed and developed for the parallel object-oriented language system pC++. In this paper, we describe the $\mathcal{T}$ design and show how programming productivity requirements for pC++ are addressed in its implementation. It is not our goal to propose $\mathcal{T}$ as a general purpose solution to parallel program analysis. Rather, our goal is to demonstrate the potential benefits of a new development strategy for program analysis tools, one that promotes meeting specific language analysis requirements over providing general purpose functionality.

The $\mathcal{T}$ program analysis environment is described in §2. The pC++ language is reviewed to establish $\mathcal{T}$'s design requirements, then the $\mathcal{T}$ architecture is presented. Some

[*]Supported in part by the National Science Foundation under grant NSF ASC92-13500, by ARPA under Rome Labs contract AF 30602-92-C-0135, by a grant from OACIS, and by IBM R&D contract MHVU3704.

[†]Department of Computer and Information Science, University of Oregon, Eugene, Oregon 97403, {*darrylb, hacks, malony, mohr*}*@cs.uoregon.edu.*

of the $\mathcal{T}$ static and dynamic analysis tools have been covered in detail elsewhere (see [9, 10]). Instead of reiterating that discussion (a brief description is given in §2.1 and §2.2), we focus on the latest $\mathcal{T}$ tool for breakpoint-based program analysis in §3. Future directions are discussed in §4.

2 The pC++ Program Analysis Environment

We advocate the development of language-specific program analysis environments. The language we are targeting to demonstrate the benefits of such an approach is pC++. pC++ is a language extension to C++ designed to allow programmers to compose distributed data structures with parallel execution semantics. The basic concept behind pC++ is the notion of a *distributed collection*, which is a type of concurrent aggregate "container class" [3]. More specifically, a *collection* is a structured set of objects which are distributed across the processing elements of the computer in a manner designed to be completely consistent with HPF [4]. To accomplish this, pC++ provides a very simple mechanism to build "collections of objects" from a base *element* class. Member functions from this element class can be applied to the entire collection (or a subset) in parallel. This mechanism provides the user with a clean interface to data-parallel style operations by simply calling member functions of the base class. In addition, there is a mechanism for encapsulating SPMD style computation in a thread-based computing model that is both efficient and completely portable. To help the programmer build collections, the pC++ language includes a library of standard collection classes that may be used (or subclassed). This includes classes such as *DistributedArray*, *DistributedMatrix*, *DistributedVector*, and *DistributedGrid*. [1]

High-level parallel languages, including pC++, place special requirements on the development and use of program analysis tools. The $\mathcal{T}$ toolset was designed to meet the following pC++ analysis requirements:

Providing a user (program-level) view. Elements of the $\mathcal{T}$ graphical interface represent objects of the pC++ programming paradigm: collections, classes, methods, and functions. These language-level objects appear in all $\mathcal{T}$ utilities.

Support for high-level, parallel programming paradigms. $\mathcal{T}$ is designed and implemented in concert with the pC++ language system. The most difficult challenge to the development of $\mathcal{T}$ is in determining what low-level performance (or debugging) instrumentation must be specified for capturing high-level execution abstractions, then translating performance data back to the application/language level.

Integration with compilers and runtime systems. $\mathcal{T}$ uses the Sage++ toolkit [1] as an interface to the pC++ compiler for instrumentation and accessing properties of program objects. $\mathcal{T}$ is also integrated with the pC++ runtime system for profiling and tracing support.

Portability, extensibility, and retargetability. Because pC++ is intended to be portable, the tools have to be portable as well. We are using C++ and C to ensure an efficient and portable implementation. The same reason led us to choose Tcl/Tk [6, 7] for the graphical interface.

[1] pC++ and its runtime system have been ported to several shared memory and distributed memory parallel systems, validating the system's goal of portability. The ports include the Kendall Square Research KSR-1, Intel Paragon, TMC CM-5, IBM SP-1, Sequent Symmetry, BBN TC2000, and homogeneous clusters of UNIX workstations using PVM; work on porting pC++ to the Cray T3D and Meiko CS-2 is in progress. More details about the pC++ language and runtime system can be found in [8, 9].

The $\mathcal{T}$ tools are implemented as graphical *hypertools*. While they are distinct tools, they act in concert as if they were a single application. Each tool implements some well-defined tasks. If one tool needs a feature of another one, it sends a message to the other tool requesting it (e.g., display the source code for a specific function). This design allows easy extensions. The Sage++ toolkit also supports Fortran-based languages, allowing $\mathcal{T}$ to be retargeted to other programming environments.

Usability. We tried to make the $\mathcal{T}$ toolset as user-friendly as possible. Many elements of the graphical user interface are analogous to *links* in *hypertext* systems: clicking on them brings up windows which describe the element in more detail. This allows the user to explore properties of the application by simply interacting with elements of most interest. The $\mathcal{T}$ tools also support *global features*. If a global feature is invoked in any of the tools, it is automatically executed in all currently running $\mathcal{T}$ tools. Examples of global features include `select-function`, `select-class`, and `switch-application`. $\mathcal{T}$ also includes a full hypertext help system.

Figure 1 shows the pC++ programming environment and the associated $\mathcal{T}$ tools architecture. The pC++ compiler frontend takes a user program and pC++ class library definitions (which provide predefined collection types) and parses them into an *abstract syntax tree* (AST). All access to the AST is done via the Sage++ library. Through command line switches, the user can choose to compile a program for profiling, tracing, and breakpoint debugging. In these cases, the instrumentor is invoked to do the necessary instrumentation in the AST. The pC++ backend transforms the AST into plain C++ with calls into the pC++ runtime system. This C++ source code is then compiled and linked by the C++ compiler on the target system. The compilation and execution of pC++ programs can be controlled by *cosy* (**CO**mpile manager **S**tatus displa**Y**). This tool provides a high-level graphical interface for setting compilation and execution parameters and selecting the parallel machine where a program will run.

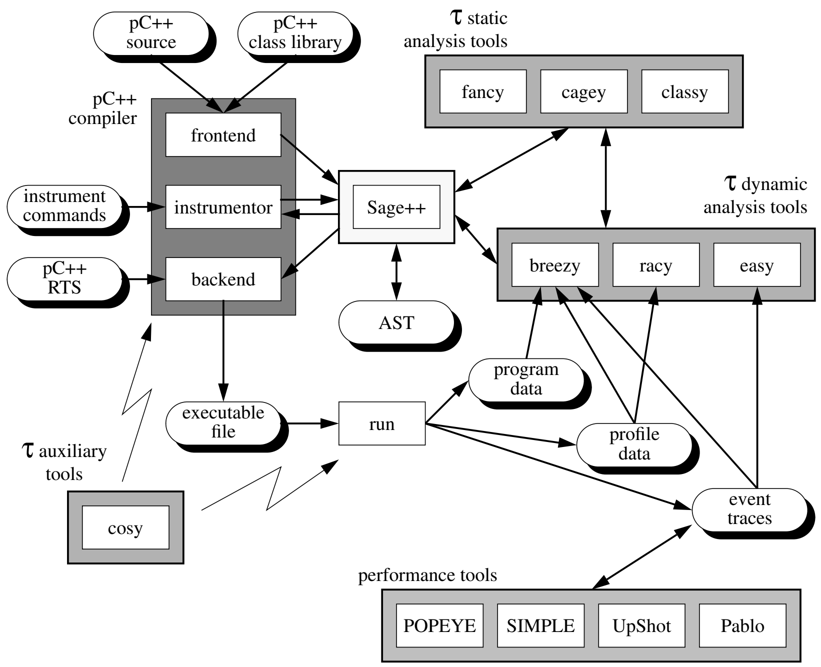

FIG. 1. *pC++ Programming Environment and $\mathcal{T}$ Tools Architecture*

The program and performance analysis environment is shown on the right side of Figure 1. They include the integrated TAU tools, profiling and tracing support, and interfaces to stand-alone performance analysis tools developed partly by other groups [11, 2, 5, 13]. The $\mathcal{T}$ toolset provides support for accessing static information about the program and for querying and analyzing dynamic data obtained from program execution. The static and dynamic tools of the $\mathcal{T}$ environment are briefly described below; a more detailed discussion of these tools can be found in [9, 10].

2.1 Static Analysis Tools

One of the basic motivations behind using C++ as the base for a new parallel language is its proven support for developing and maintaining complex and large applications. However, to apply the C++ language capabilities effectively, users require support tools to manage and access source code at the level of programming abstractions. This is even more important for pC++ because of its data-parallel programming abstractions and SPMD execution paradigm.

Currently, $\mathcal{T}$ provides three tools to enable the user to quickly get an overview of a large pC++ program and to navigate through it: the global function and method browser *fancy* (**F**ile **AN**d **C**lass displa**Y**), the static callgraph display *cagey* (**CA**ll **G**raph **E**xtended displa**Y**), and the class hierarchy display *classy* (**CLASS** hierarch**Y** browser). The tools are integrated with the dynamic analysis tools through the global features of $\mathcal{T}$, allowing the user to easily find execution information about language objects. For instance, to locate the corresponding dynamic results (after a measurement has been made), the user only has to click on the object of interest (e.g., a function name in the callgraph display).

2.2 Dynamic Analysis Tools

Dynamic program analysis tools allow the user to explore and analyze program execution behavior. This can be done in three general ways. *Profiling* computes statistical information to summarize program behavior, allowing the user to find and focus quickly on the main bottlenecks of the parallel application. *Tracing* portrays the execution as a sequence of abstract *events* that can be used to determine various properties of time-based behavior. *Breakpoint debugging* allows a user to stop the program at selected points and query the contents of program state. For all analysis modes, the most critical factor for the user is how the high-level program semantics are related to the measurement results. $\mathcal{T}$ helps in presenting the results in terms of pC++ language objects and in supporting global features that allow the user to locate the corresponding routine in the callgraph or source text by simply clicking on the related measurement result or state objects.

$\mathcal{T}$'s dynamic tools currently include an execution profile data browser called *racy* (**R**outine and data **AC**cess profile displa**Y**), an event trace browser called *easy* (**E**vent **A**nd **S**tate displa**Y**), and a breakpoint debugger called *breezy*. To generate the dynamic execution data for these tools, profiling and tracing instrumentation and measurement support has been implemented in pC++.

In Figure 2 we show how the $\mathcal{T}$ tools are used in combination on the Poisson code of the pC++ benchmark suite. After compiling and executing the benchmark using 16 nodes on our Sequent Symmetry with *cosy*, the individual tools were launched using the $\mathcal{T}$ main control panel. Then the global feature `select-class` called from the *breezy* main control panel (see Figure 4) was used to show the `vector` class declaration in the file viewer window of *fancy* (lower right corner), its members in the member table window of *classy* (upper

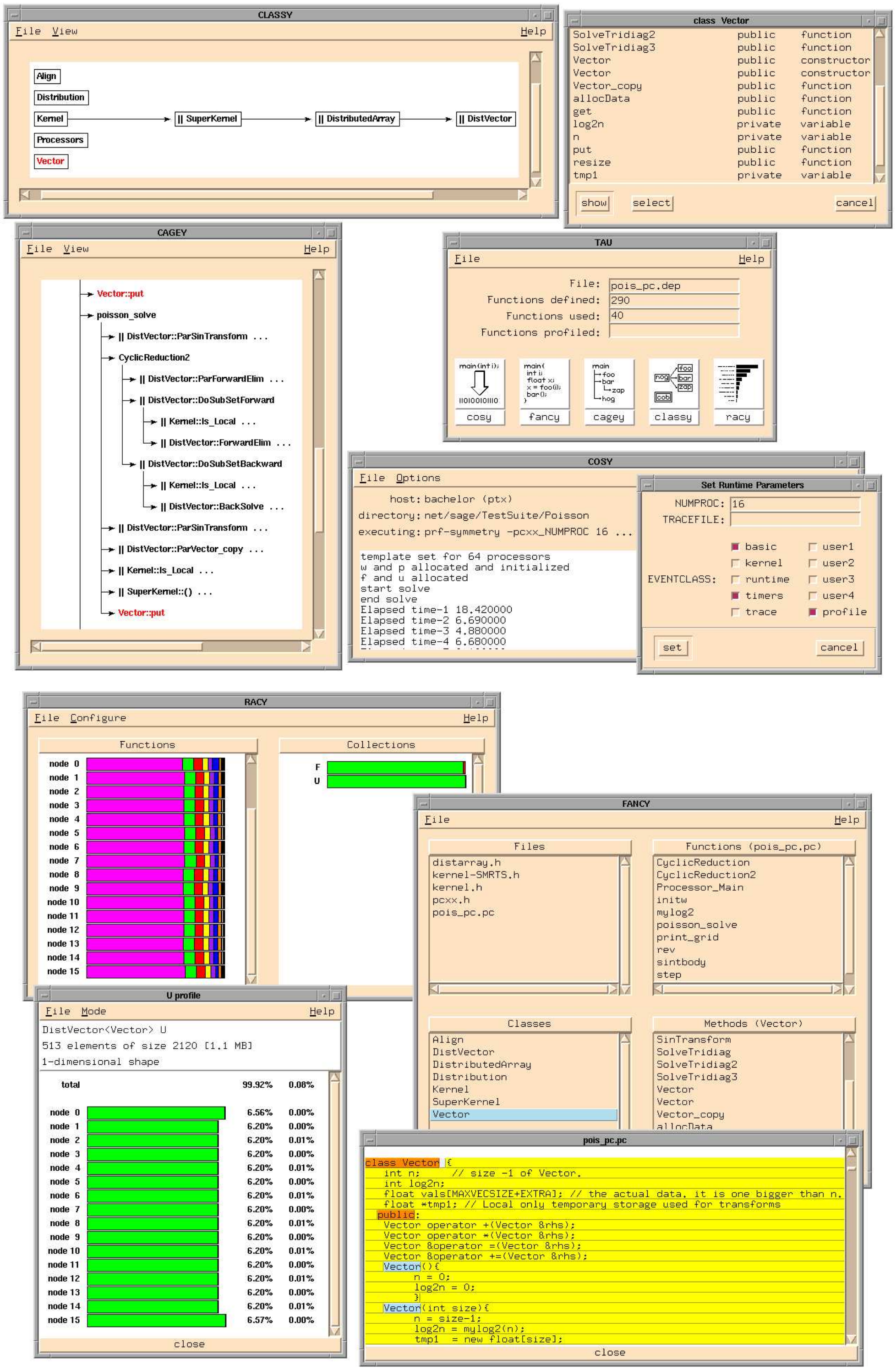

FIG. 2. $\mathcal{T}$ *Tools*

right corner), and data access statistics from *racy* (lower left corner). *Racy* uses colored bars and two columns of numbers to show the percentage of local and remote accesses to this collection.

3 A $\mathcal{T}$ Tool for Breakpoint-Based Program Analysis

The most recently developed $\mathcal{T}$ tool, *breezy* (**BR**eakpoint **E**xecutive **E**nvironment for visuali**Z**ation and data displa**Y**) is for breakpoint-based program analysis. In addition to describing the functionality of *breezy* in detail, we use the discussion below to show how a tool is integrated into the $\mathcal{T}$ environment and interacts with the other program analysis tools. *Breezy* incorporates many of the features of the $\mathcal{T}$ environment and is representative of the level of sophisticated functionality that $\mathcal{T}$ supports and future $\mathcal{T}$ tools will provide.

3.1 *Breezy* Design

Breezy allows a user to control the execution of a pC++ program and to view the parallel program collection data during the execution, the essential functions of a parallel debugger. The execution is controlled by manipulating program breakpoints. At these breakpoints, the program data can be displayed in a text window or visualized by a compatible visualization tool. The architecture of *breezy* is shown in Figure 3.

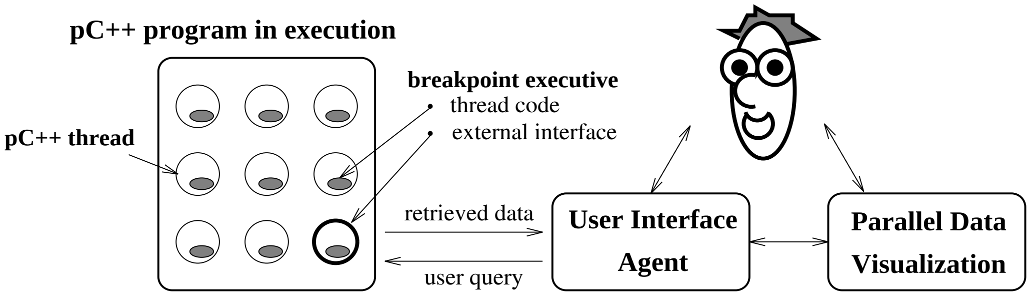

FIG. 3. Breezy *Architecture*

There are two principal issues to address in the *breezy* design: breakpoint implementation and program state access. In general, supporting breakpoints for debugging parallel programs is difficult because consistent global program states are difficult to reconstruct from parallel execution. pC++'s data-parallel SPMD-style of computation reduces the complexity of the problem by restricting the possible execution behaviors to single-level fork-join type. In addition, we constrain breakpoints to occur only within barriers. It may seem restrictive to allow access only at barriers, but a pC++ computation usually involves many barriers, and this restriction is not too severe for our purposes of viewing parallel program states.

At each breakpoint, all machine nodes enter a barrier except for a master node, the *breakpoint executive*. This node establishes an interface between the *breezy user interface agent* and the distributed collection data (see Figure 3). The other nodes act as slaves to the breakpoint executive while waiting in this barrier. The user can enter high-level requests for collection data to the interface agent, which forwards them to the breakpoint executive node. This node collects the appropriate data from all other nodes and returns it to the interface agent. When the user specifies to continue to the next breakpoint, the breakpoint executive node finally joins all others in the barrier, consequently allowing all to continue until the next breakpoint.

Since pC++ runs on a range of parallel machines, each one having different symbol table and binary formats, gaining access to internal program variables in a portable way is a challenge. The approach we took was to implement *breezy* at the language level, utilizing data-parallel program semantics and Sage++ program transformation support.

Breezy focuses on providing access to the distributed collections of a pC++ program. During program compilation (see Figure 1), *breezy* automatically generates access functions and methods for all the parallel data structures. Using Sage++ routines, the parallel data structures are easily located in the program and access functions (or access methods if the structure is a class) are inserted appropriately in the program code. The modified program is then linked with a special instrumented runtime system containing code to keep a record of collection instantiations. This record is subsequently used to build a collection symbol table during runtime. The runtime system also includes the modified barrier function that calls the breakpoint executive code. Note that all program modifications are at the source code level, implemented in the pC++ language itself. The obvious benefit is that this additional code is Portable to everywhere pC++ runs. The less obvious benefit is that the operations to access the parallel data are "generated" as a natural result of compiling the instrumented program.

3.2 *Breezy* Operation

Using *breezy*, the execution of a pC++ program proceeds in steps. At breakpoint boundaries, the application relinquishes control to the breakpoint executive. The user then interacts with the breakpoint executive through the external *breezy* interface agent to access the program state. Once finished, the user instructs the executive to continue to the next breakpoint or to the end of the program. During an initial barrier, the executive connects to the interface agent via an Internet domain TCP/IP socket. This method of control allows a user to manipulate the program and its data from any machine that is accessible by the parallel host on which the pC++ application is actually executing.

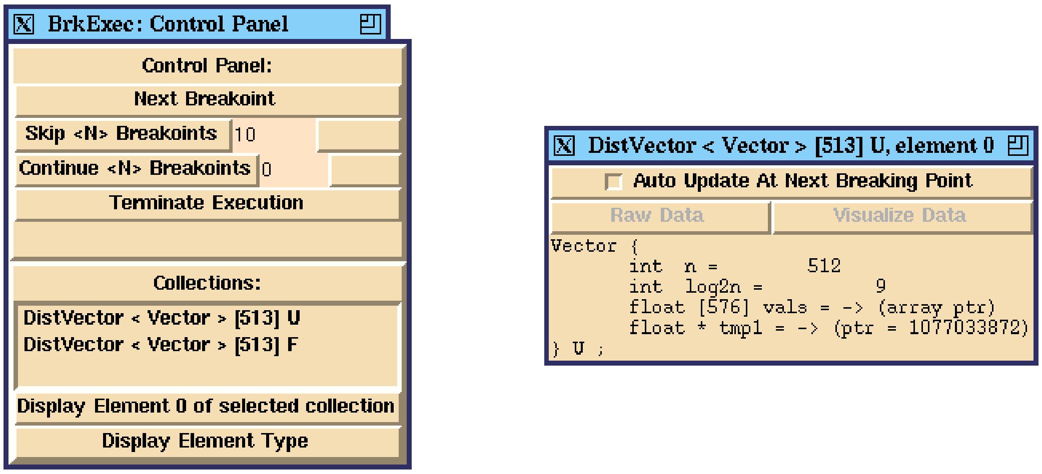

FIG. 4. Breezy *Main Control Panel (left) and Collection Element Display (right)*

The *breezy* interface agent offers a main panel (see Figure 4, left) for controlling program execution and listing currently active collections that are available to be accessed. From this panel, the user has control of when to stop and what data to examine. Currently, to control the program execution, the user can step from each breakpoint to the next, specify a number of breakpoints to skip before again pausing to examine the data, or continue or terminate the parallel application.

To access collection data from the main control panel, the user can specify a particular collection instance of interest by selecting a collection in the control panel where all collections are listed. Collection structure information comes from the program's AST and is access via Sage++. This functionality was easily implemented as a result of *breezy*'s integration in the $\mathcal{T}$ environment.

For each collection that the user selects, a new window appears, displaying the name of the collection element structure, each field name, the values of each field from the zero-th element of the collection, and the name of the collection (see Figure 4, right). The output is formatted to look like a C++ style declaration. Selecting the *Auto Update* toggle button causes the data values to be updated with current information at each breakpoint. The remaining two buttons are for viewing raw data and visualizing the data. These last two buttons work in concert with the display of the element structure. Any one of the fields of the displayed structure can be selected with the mouse. This selection specifies which field is to be viewed or visualized. Thus, until a selection is made, the two buttons are disabled.

The user can retrieve raw data from a structure as text, which is put into a scrollable window for viewing. This window contains the automatic update toggle button which works as described above.

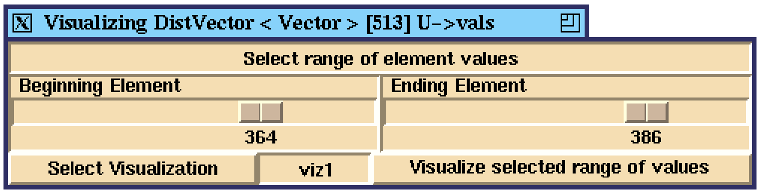

FIG. 5. Breezy *Visualization Control Window*

The user can also visualize the parallel data retrieved. When a user chooses to visualize a field of a structure, a new window is created (see Figure 5). The user must specify the range of elements to retrieve from the collection instance. Also, the user must select a visualization from a menu of the provided (and user-defined) visualizations. Currently, a list of basic visualizations are supplied: triangle mesh, sphere mesh, color mesh, and cube mesh. After these user specifications, the visualization can be started. Separate processes are spawned from the interface to accomplish the visualizations, and then data is piped into them. The visualization performs the mapping of data to graphical objects, producing a visualization/animation of the collection data (see Figure 6). After starting a visualization, a small interface window remains behind, informing the user about the visualization, and again offering the automatic update toggle button. As in other windows, if this button is selected the visualization will be sent current information at each breakpoint, thus supplying the data for an animation during the execution of the program. Through the global features of $\mathcal{T}$, the user can also view the corresponding declarations (in *fancy*) or the access profiles (in *racy*) of collection classes that were selected for visualization (see Figure 2).

Several improvements will be implemented in the near future. The current prototype lets the user select a visualization from a predefined list. Future versions of *breezy* will allow a more flexible mapping through a (perhaps graphical) *mapping specification*. We also plan to enhance the breakpoint executive to allow access to the profile and trace data buffers, and automatically update the *racy* and *easy* displays during runtime, thus allowing for on-line performance analysis. We are also involved in another project which has developed a new visualization prototyping environment called *POPEYE* (**P**arallel pr**O**gram and **P**erformance displa**Y** **E**nvironment). *POPEYE* is an external performance tool that interacts with $\mathcal{T}$ through profile and trace data capable of creating sophisticated, multi-dimensional

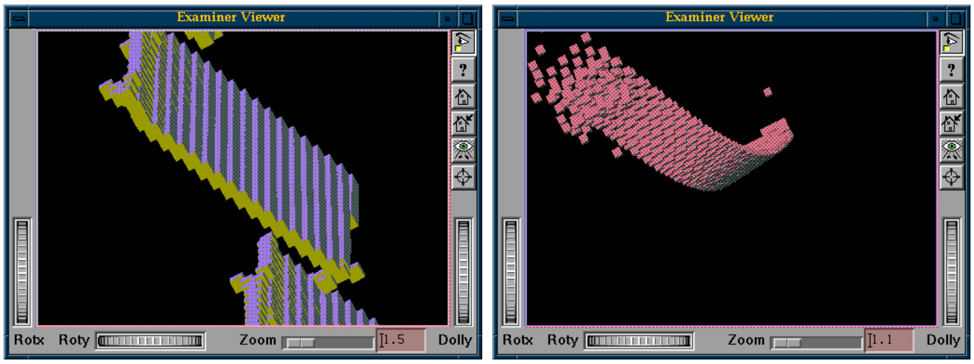

FIG. 6. Breezy *Visualizations*

visualizations. Some example views are shown in Figure 7. We are working on reusing the *POPEYE* tools as visualization backends for *breezy*. More details about *POPEYE* can be found in [11, 12].

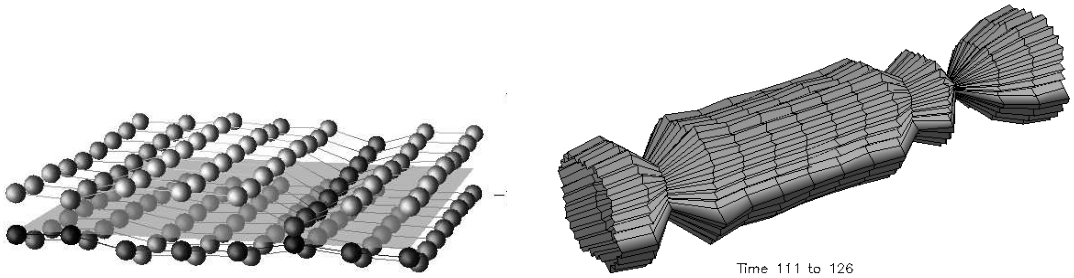

FIG. 7. POPEYE *pC++ performance visualizations: Local/Remote Accesses to Matrix collection (left) and remote access ratios as Kiviat tube (right)*

Another enhancement we plan on soon implementing is allowing users to provide custom access functions in their parallel data structures, accessible through the breakpoint executive. In this way, the user can perform some preliminary analysis, grouping, and filtering of the data before it is sent over the socket connection, possibly saving costly network communication overhead. Another benefit of customizing data retrieval is the fact that it can be tailored for specific visualizations also provided by the user. Also, the customized access functions could modify the data itself, providing the basis for computational steering.

4 Conclusion and Future Work

The pC++ programming system includes an integrated set of performance instrumentation, measurement, analysis, and visualization tools. With this support, we have been able to validate performance scalability claims and characterize important performance factors of the runtime system ports during pC++ system development [9]. As a consequence, the first version of the compiler is being introduced with an extensive set of performance experiments already documented. More importantly, the $\mathcal{T}$ environment demonstrates the advantages of language-specific program analysis tools. Its design and integration with the pC++ language system has enabled more sophisticated debugging and performance analysis tools to be developed and effectively applied than would have been possible by retrofitting general-purpose toolkits. In consequence, we believe this will allow important programming productivity issues to be better addressed in the pC++ language environment.

The pC++ project is an on-going research effort. We are planning to enhance the current version in several ways. First, the pC++ language will be extended to include task-parallel programming and more aggressive compiler optimizations. This will complicate program and performance analysis, requiring a new generation of tools to be developed. Second, we are working on enhanced tools for the programming environment: better graphical instrumentation control, integration of data distribution visualizations, full debugger, etc. Lastly, we intend to provide interaction with other high-performance languages like HPF, CC++ , and Fortran-M.

For more information ...
Documentation and source code for pC++ and Sage++ are available via anonymous FTP from `ftp.cs.indiana.edu` or `cica.cica.indiana.edu` in the directory `~ftp/pub/sage`, or via WWW at `http://www.cica.indiana.edu/sage`. $\mathcal{T}$ is part of the pC++ distribution.

References

[1] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, B. Winnicka, *Sage++: An Object Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools*, Proc. Oonski '94, Oregon, 1994.

[2] V. Herrarte, E. Lusk, *Studying Parallel Program Behavior with Upshot*, Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[3] A. Chien and W. Dally. *Concurrent Aggregates (CA)*, Proc. 2nd ACM Sigplan Symposium on Principles & Practice of Parallel Programming, Seattle, Washington, March, 1990.

[4] High Performance Fortran Forum, *High Performance Fortran Language Specification*, 1993. Available from titan.cs.rice.edu by anonymous ftp.

[5] B. Mohr, *Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible?*, Proc. CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing, St. Hilaire du Touvet, France, Elsevier, Advances in Parallel Computing, Vol. 6, pp. 103–124, 1993.

[6] J. Ousterhout, *Tcl: An Embeddable Command Language*, Proc. 1990 Winter USENIX Conf.

[7] J. Ousterhout, *An X11 Toolkit Based on the Tcl Language*, Proc. 1991 Winter USENIX Conf.

[8] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, B. Mohr, *Implementing a Parallel C++ Runtime System for Scalable Parallel Systems*, Proc. 1993 Supercomputing Conference, Portland, Oregon, pp. 588–597, Nov. 1993.

[9] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, *Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers*, Proc. 8th Int. Parallel Processing Symb. (IPPS), Cancún, Mexico, IEEE Computer Society Press, pp. 75–85, Apr. 1994.

[10] B. Mohr, D. Brown, A. Malony, *TAU: A Portable Parallel Program Analysis Environment for pC++* , Proc. of CONPAR 94 - VAPP VI, Linz, Austria, Springer Verlag, LNCS, Sept. 1994.

[11] S. Hackstadt, A. Malony, *Next-Generation Parallel Performance Visualization: A Prototyping Environment for Visualization Development*, Proc. Parallel Architectures and Languages Europe, (PARLE), Athens, Greece, 1994.

[12] S. Hackstadt, A. Malony, B. Mohr, *Scalable Performance Visualization for Data-Parallel Programs*, Proc. Scalable High Performance Computing Conf. (SHPCC), Tennessee, 1994.

[13] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyasta, T. Birkett, D. W. Jensen, B. A. A. Nazief, B. K. Totty, *Scalable Performance Environments for Parallel Systems.* Proc. 6th Distributed Memory Computing Conference, IEEE Computer Society Press, pp. 562–569, 1991.