Submitted to: CONPAR 94 - VAPP VI, University of Linz, Austria, September 6-8, 1994.

TAU: A Portable Parallel Program Analysis Environment for pC++ [1]

Bernd Mohr, Darryl Brown, Allen Malony
Department of Computer and Information Science
University of Oregon, Eugene, Oregon 97403
{*mohr, darrylb, malony*}*@cs.uoregon.edu*

Abstract

The realization of parallel language systems that offer high-level programming paradigms to reduce the complexity of application development, scalable runtime mechanisms to support variable size problem sets, and portable compiler platforms to provide access to multiple parallel architectures, places additional demands on the tools for program development and analysis. The need for integration of these tools into a comprehensive programming environment is even more pronounced and will require more sophisticated use of the language system technology (i.e., compiler and runtime system). Furthermore, the environment requirements of high-level support for the programmer, large-scale applications, and portable access to diverse machines also apply to the program analysis tools.

In this paper, we discuss $\mathcal{T}$ (TAU, **T**uning and **A**nalysis **U**tilities), a first prototype for an integrated and portable program analysis environment for pC++, a parallel object-oriented language system. $\mathcal{T}$ is integrated with the pC++ system in that it relies heavily on compiler and transformation tools (specifically, the Sage++ toolkit) for its implementation. The paper describes the design and functionality of $\mathcal{T}$ and shows its application in practice.

Keywords: parallel programming environments, parallel languages, integrated program analysis tools, profiling, event tracing

1 Introduction

The unavailability of comprehensive, user-oriented programming environments is one of the foremost obstacles to the routine application of parallel, high performance computing technology. Most critical is the need for advances in integrated parallel debugging, performance evaluation, and program visualization tools [2]. Current tools fail to adequately address parallel programming productivity requirements for several reasons:

[1] This research is supported by DARPA under Rome Labs contract AF 30602-92-C-0135.

(R1) *Providing a user (program-level) view.* Tool development has been dominated by efforts directed at the execution level (e.g., efficient implementation of monitoring [1]). In consequence, tool users are given little support for "translating" program-level semantics to and from low-level, execution measurements and runtime data.

(R2) *Support for high-level, parallel programming languages.* The development of advanced parallel languages (e.g., HPF [6] and pC++ [14]) further separates the user from execution-time reality, because of the complex transformations and optimizations that take place between the layers of language abstraction, runtime paradigm, and execution environment.

(R3) *Integration with compilers and runtime systems.* The majority of debugging and performance analysis tools have been developed independent of parallel languages and runtime systems, resulting in poor reuse of base-level technology, incompatibilities in tool functionality, and interface inconsistencies in the user environment.

(R4) *Portability, extensibility, retargetability.* Users of portable languages should be provided a consistent program development and analysis environment across multiple execution platforms. The tools should be extensible, so their functionality can be increased to accommodate new language or runtime system features. Support for retargetability allows the tool design to be easily reused for different parallel languages and system environments.

(R5) *Usability.* Implementing a high-level, portable, integrated tool does not automatically result in an easy-to-use tool. In the past, less emphasis was put on well-designed interfaces which led to very powerful but poorly used program analysis tools.

These failures will become even more significant as we move towards more general and robust high performance parallel languages with highly optimized runtime systems.

Practically, the problems are ones of tool *design technology* rather than *functionality*: existing tools provide a variety of functionality, but they have not been successfully integrated into usable parallel programming environments. One approach to improving integration is to base the design of tools on the particular performance and debugging requirements of the parallel language system in which the tools will be used. In this manner, tool functionality can specifically target program analysis support where tool application is well understood. However, unless tool implementation can leverage other programming system technology (e.g., the simple use of the compiler to implement instrumentation), the integration of the tools in the environment cannot be fully realized.

We have designed and developed a parallel program analysis environment, $\mathcal{T}$ (TAU, **T**uning and **A**nalysis **U**tilities), for a 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. Our goal is not to propose $\mathcal{T}$ itself 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 analysis requirements over providing general purpose functionality.

The pC++ language system and the Sage++ restructuring toolkit that forms the basis of the pC++ compiler are briefly described in §2. From this description, we develop a general model of pC++ program observability and analysis. The $\mathcal{T}$ environment for pC++, presented in §3, is based on this model and reflects the critical program analysis requirements of the pC++ language. We outline the capabilities of the $\mathcal{T}$ tools, showing specific instances of their application and explaining how they are implemented in the pC++ programming environment.

2 A Brief Introduction to 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" [5, 7]. 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 Fortran [6]. To accomplish this, pC++ provides a very simple mechanism to build "collections of objects" from some 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*.

In its current form, the pC++ compiler is a preprocessor that generates C++ code and machine-independent calls to a portable runtime system. This is accomplished by using the object-oriented compiler preprocessor toolkit Sage++ [3]. It provides the functions necessary to read and restructure an internal representation of the pC++ program. After restructuring, the program is "unparsed" back into C++ code, which can be compiled on the target architecture and linked with a runtime system specifically designed for that machine (see Figure 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, 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, 14, 15].

2.1 An Observability and Analysis Model for pC++

Collection definition and use are the key aspects for program analysis in the pC++ system. Programmers using pC++ require support for observing the collection data structures with respect to both their object-oriented definition and their parallel execution semantics. Collection observation can be static and dynamic. In each case, a high-level language view is needed because the programmer should not have to know or understand the effects of pC++ program transformations. The program analysis tools, on the other hand, do need to be able to access various information associated with a pC++ program, its compilation, and its execution. This data must be made consistent with the user's view of the pC++ programming paradigm, and with aspects of the program development and evaluation process in which the user is involved. For example, we can assume that the programmer is aware that collection elements are operated upon by multiple processors, but may not understand how the runtime system supports element access. Hence, when collection performance data is presented to the user, it should be with respect to collection definition and execution semantics (e.g., by collection element references and total collection method execution times). Access to low-level communication timing information, for instance, is not available as a default. Simply put, the pC++ analysis model that $\mathcal{T}$ supports is based entirely on language and execution semantics, leaving the task of associating static and dynamic data at multiple levels in the language system to tool implementation in the programming environment.

3 The pC++ Program Analysis Environment

In this section, we discuss $\mathcal{T}$ (TAU, **T**uning and **A**nalysis **U**tilities)[2], a first prototype for an integrated portable pC++ program and performance analysis environment. The $\mathcal{T}$ toolset was designed to meet the requirements described in §1.

(R1) *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.

(R2) *Support for high-level, parallel programming languages.* $\mathcal{T}$ is unique because it is defined by the program analysis requirements of the pC++ language. Also, $\mathcal{T}$ is designed

[2] Internally, TAU is fondly referred to as **T**ools **A**re **U**s.

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.

(R3) *Integration with compilers and runtime systems.* $\mathcal{T}$ uses the Sage++ toolkit 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.

(R4) *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, portable, and reusable implementation. The same reason led us to choose Tcl/Tk [12, 13] for the graphical interface.

The $\mathcal{T}$ tools are implemented as graphical *hypertools*. While they are distinct tools, they act in concert like they were one single application. Each tool implements some 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.

(R5) *Usability.* We tried to make the $\mathcal{T}$ toolset as user-friendly as possible. Many elements of the graphical user interface act like *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.

3.1 Tool Architecture Overview

Figure 1 shows an overview of the pC++ programming environment. 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 a command line switch, the user can choose to compile a program for profiling or tracing. In both cases, the instrumentor is invoked to do the necessary instrumentation in the AST (see Section 5.1 for details). 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.

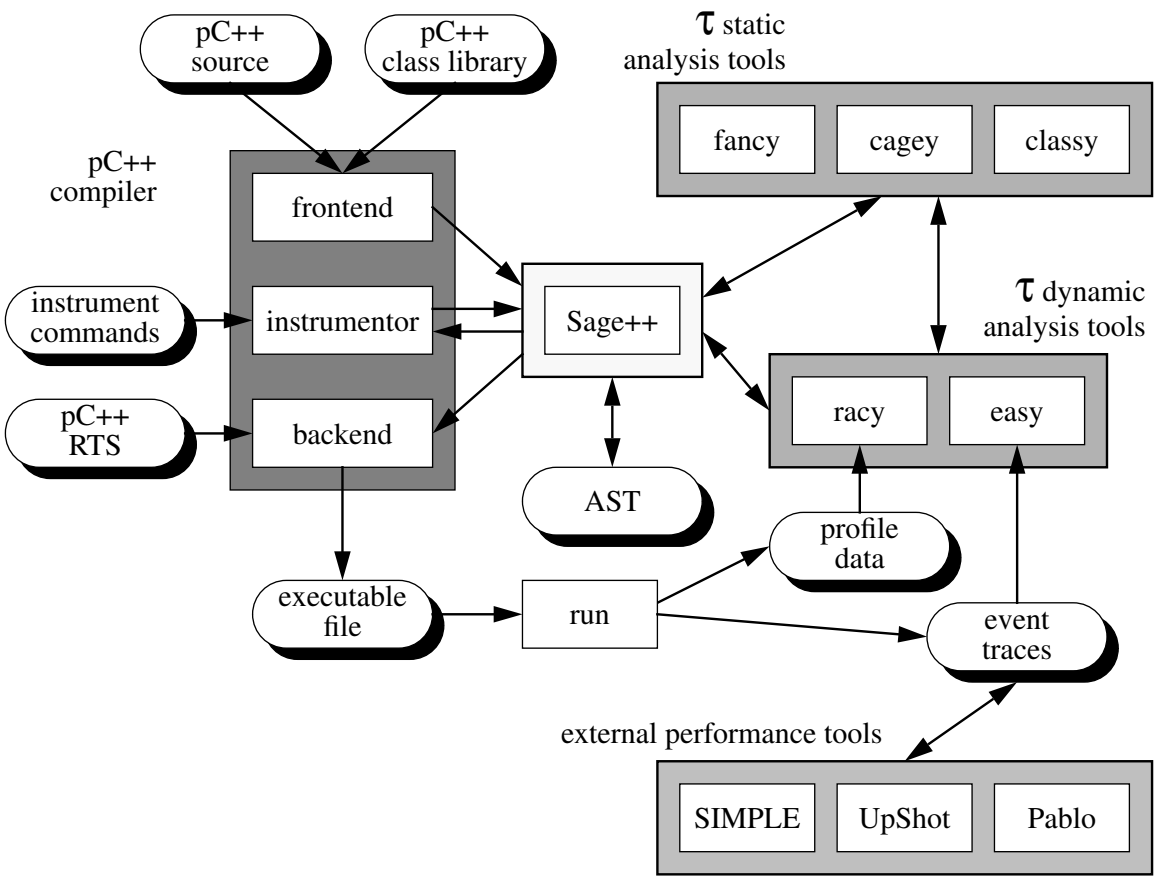

Figure 1: $\mathcal{T}$ Tools Architecture

The program and performance analysis environment is shown on the right side. They include the TAU tools, profiling and tracing support, and interfaces to performance analysis tools developed by other groups [4, 10, 11, 16, 17]. In the following, the $\mathcal{T}$ static and dynamic tools are described in more detail.

4 Static Analysis Tools

One of the basic motivations behind using C++ as a 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++.

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: a global function and method browser (*fancy*), a static callgraph display (*cagey*), and a class hierarchy display (*classy*). In addition, the tools allow the user to easily find execution information about language objects as they are

integrated with the dynamic analysis tools through the global features of $\mathcal{T}$. 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).

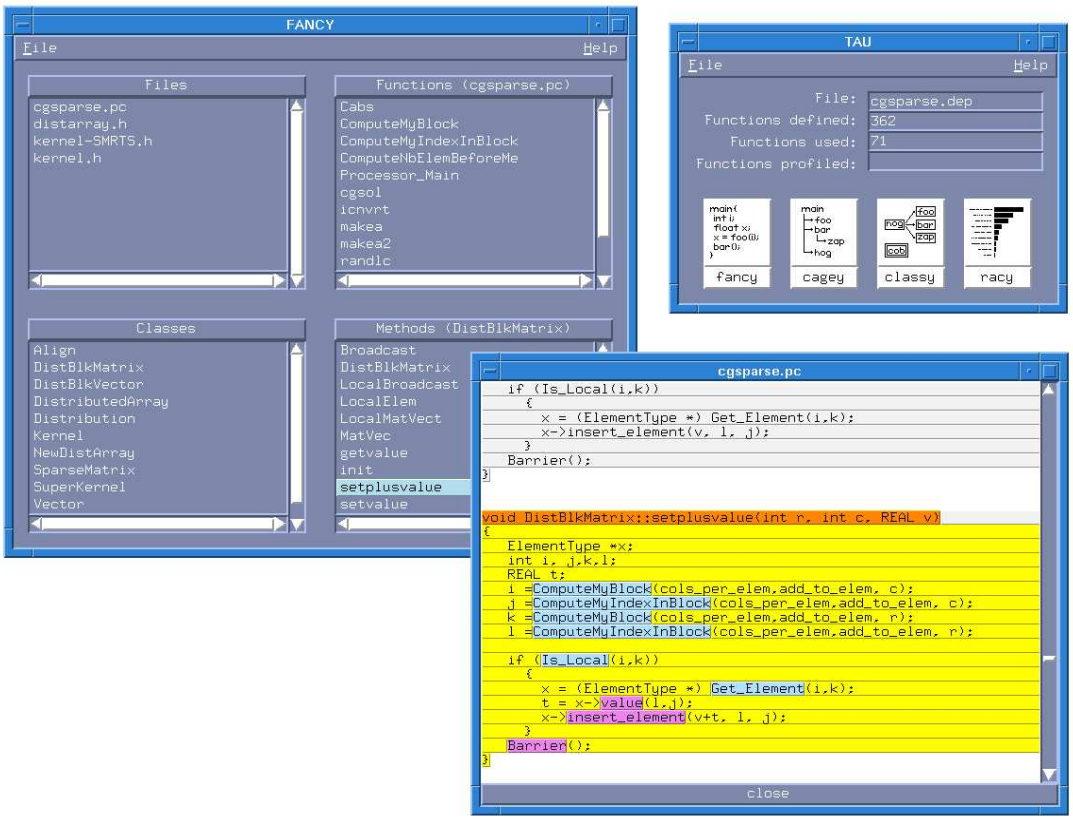

Figure 2: $\mathcal{T}$ Master Control Window and Fancy

4.1 Fancy

Fancy (**F**ile **AN**d **C**lass displa**Y**) lets the user browse through the files and classes that compose the application, allowing the source text of its functions or methods to be quickly located. The *main window* displays four listboxes (see Figure 2). The two on the left show all source files used (*Files*) and all classes defined (*Classes*) for the current application. Selecting one item in either of these listboxes displays all global functions defined for the selected file (in *Functions*) or all methods of the selected class (in *Methods*), respectively.

Selecting a routine (either a global function or class method) displays the source text of the selected procedure in a separate viewer window. In Figure 2 the method `setplusvalue` of class `DistBlkMatrix` was selected. The header and body of the currently selected routine as well as its children (functions and methods which are called from that routine) are highlighted

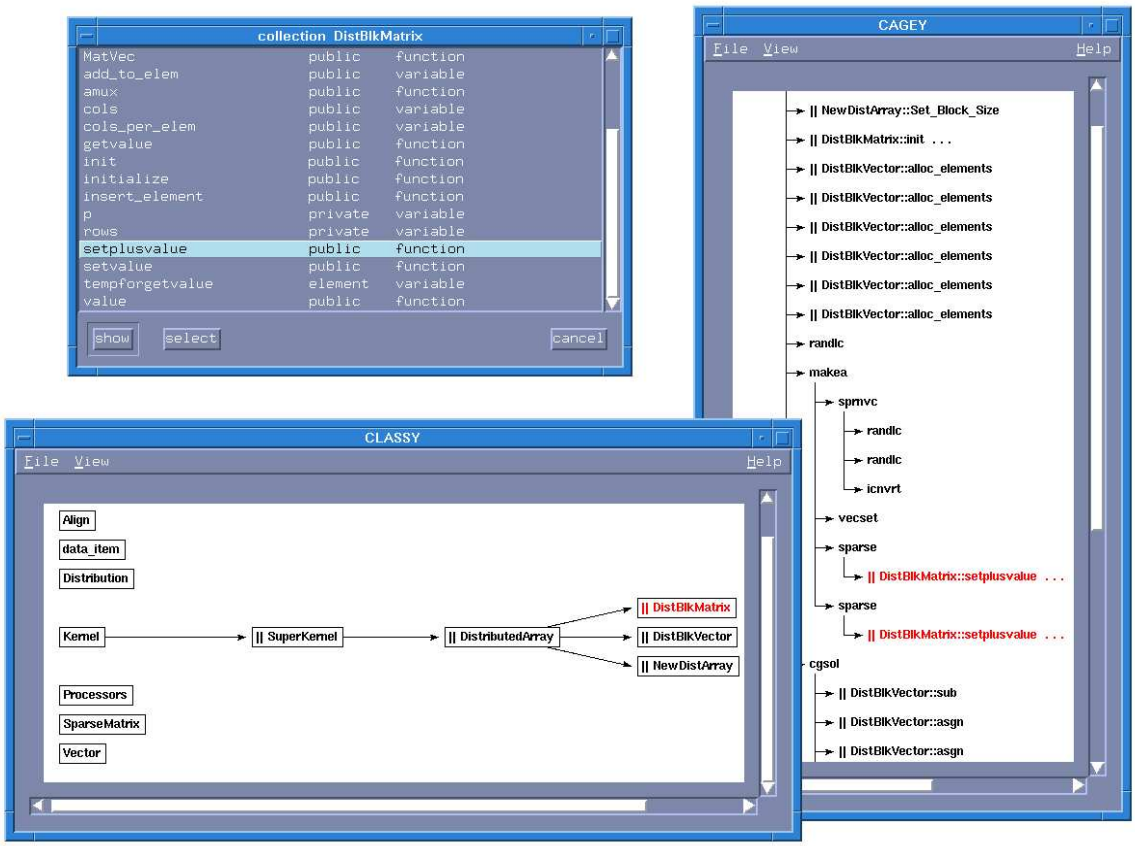

Figure 3: Cagey and Classy

using different colors. Routines can also be selected by clicking on the children in the viewer window. Class definitions can be displayed by selecting the class name in the function header. In this manner, *fancy* provides high-level access to pC++ program structure and source code.

4.2 Cagey

Cagey (**CA**ll **G**raph **E**xtended displa**Y**) shows the static callgraph of the functions and methods of the current user application (see Figure 3). It uses Sage++ to determine the callgraph structure and to differentiate between global functions and class methods. *Cagey* enables the user to quickly observe the calling structure of the program and to locate those parts where parallelism is involved by marking routines which are executed in parallel with the string "||" before the name.

As the callgraph can be quite big for large applications, *cagey* allows one to display each routine in two modes: *show* or *hide* children. If a routine has children but is in *hide children* mode, a "..." is displayed behind the name of the function. Clicking on a function toggles its mode. If a function is "expanded" the first time, only the direct children are shown.

To invoke the global feature `select-function`, the user simply clicks on a function name.

Within *cagey*, this results in highlighting all appearances of the currently selected function in the callgraph by showing them in red (e.g., `DistBlkMatrix::setplusvalue` in Figure 3).

4.3 Classy

Classy (**CLASS** hierarch**Y** browser) is a class hierarchy browser for programs written in C++ and languages based on C++ like pC++. Classes which have no base class (called level 0 classes) are shown in a column on the left side of the display window (see Figure 3). Subclasses derived from level 0 classes are shown in the next column to the right and so on.

As in *cagey*, *classy* lets the user choose the level of detail in the class hierarchy display by allowing him to fold or expand subtrees in the graph. If a class has subclasses but was folded into one node, the name of the class is shown within a double line border. In addition to showing the class relationships, *classy* allows quick access to key properties of a class. Collections are marked with a "||" before the name. By selecting a class with the right mouse button, the *member table window* is displayed, showing a detailed list of all its members (as it is for the collection `DistBlkMatrix` in Figure 3), which includes

- the field name,
- whether this field is a *public*, *private*, *protected*, or method of *element* class member
- whether this field is a *variable*, *function*, *operator*, *constructor*, or *destructor*
- whether the field was declared *inline* and/or *virtual*

Also, if *fancy* is running, the source code of the class definition is shown in its viewer window, by using the global feature `select-class`. Also, clicking on a function member in the member table selects it. Providing an application browser based on classes (objects) and their relationships (inheritance), *classy* directly supports object-oriented design and programming in pC++.

5 Dynamic Analysis Tools

Dynamic program analysis tools allow the user to explore and analyze program execution behavior. This can be done in two 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 behavior as a sequence of abstract *events* that can be used to determine various properties of time-based behavior.

The most critical factor for the user is to relate the measurement results back to the source code. $\mathcal{T}$ helps in presenting the results in terms of pC++ language objects and in

supporting global features that allows the user to locate the corresponding routine in the callgraph or source text by simply clicking on the related measurement result objects.

Before presenting the $\mathcal{T}$ dynamic analysis tools *racy* and *easy*, we briefly describe the approach used to implement profiling and tracing in pC++ (see [15] for more details).

5.1 Portable Profiling for C++ and Languages based on C++

A very valuable tool for program tuning is function profiling. Here, special instrumentation code is inserted at all entry and exit points of each function to capture data that can be used to calculate the number of times this function is called and the percentage of the total execution time spent there.

To ensure portability, all instrumentation for profiling must be done at the source language level. Using language features of C++, we can instrument the source code efficiently, just by declaring a special *Profiler* class which only has a constructor and a destructor and no other methods. A variable of that class is then declared in the first line of each function which has to be profiled. During runtime, a `Profiler` object is created and initialized each time the control flow reaches its definition (via the constructor) and destroyed on exit from its block (via the destructor).

This generic profiling instrumentation approach has two basic advantages. First, the instrumentation is *portable*, because it occurs at the source code level. Second, different implementations of the profiler can be easily created by providing different code for the constructor and destructor. This makes it very *flexible*. Currently, we have implemented two versions of the profiler for pC++:

Direct Profiling: The function profile is directly computed during program execution. For each profiled function we compute the number of calls and the running sum of time used by this function including and excluding its children.

Trace-based Profiling: Here the constructor and destructor functions simply call an event logging function from the pC++ software event tracing library (see Section 5.3). The computation of the profile statistics is then done off-line.

Other profiling alternatives could be implemented in the same way. For example, profiling code could be activated/deactivated for each function separately, allowing dynamic profiling control. Another possibility is to let users supply function-specific profile code (specified by source code annotations or special class members with predefined names) that allows customized runtime performance analysis.

We use the Sage++ class library and restructuring toolkit to manipulate pC++ programs and insert the necessary profiler instrumentation code at the beginning of each function. The user has control over the level, detail, and type of profiling.

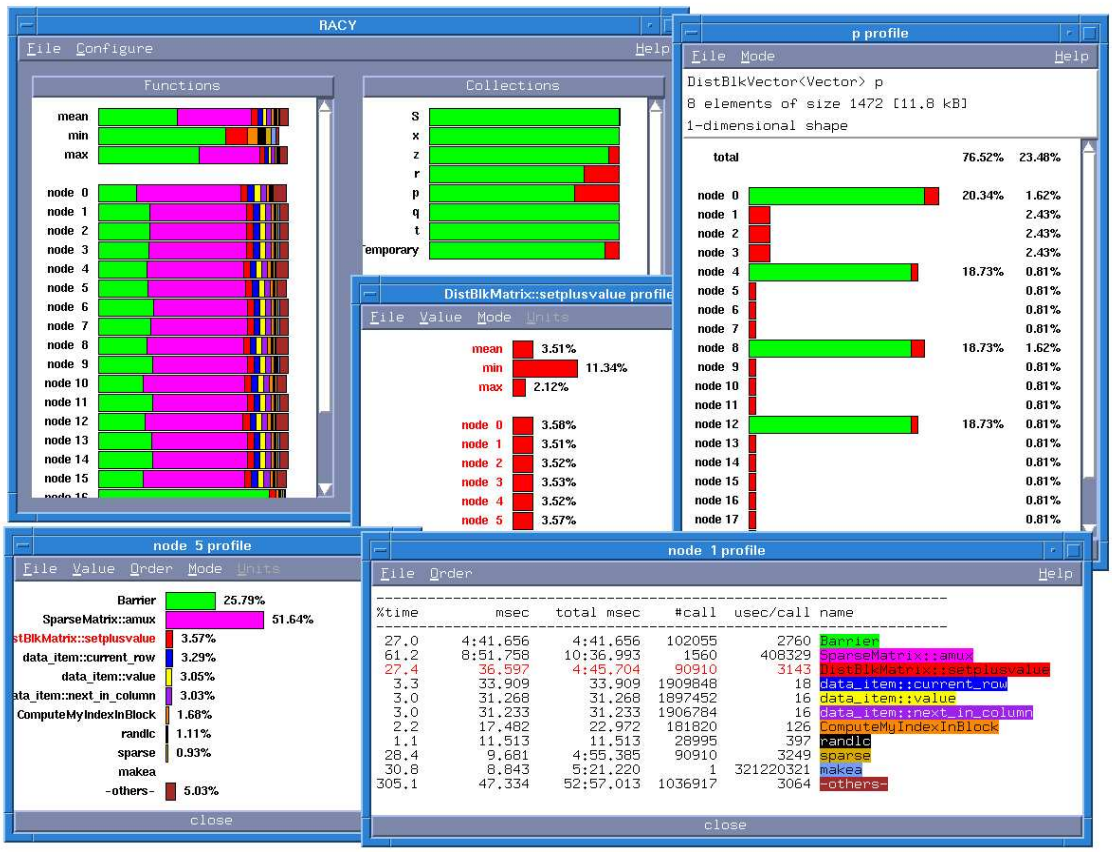

Figure 4: Racy

There are also instrumented versions of the pC++ class libraries and runtime system, both for direct and trace-based profiling. In addition to the instrumentation of user-level functions, they provide profiling of runtime system functions and collection access.

5.2 Racy

Racy (**R**outine and data **AC**cess profile displa**Y**) is the $\mathcal{T}$ *parallel profile data viewer*. After compiling an application for profiling and running it, *racy* lets you browse through the function and collection access profile data generated. As with the other $\mathcal{T}$ tools, *racy* lets the user choose the level of detail of the data displayed. The main window (see Figure 4) gives a quick overview of the execution of the application by summarizing function (left) and collection access (right) performance in two graphs.

The *function profile summary* presents, for each node the program was running on, a single bargraph line showing the percentage of total runtime the program spent in specific functions. In addition, the mean, maximum, and minimum values are shown on top of the graph. *Racy* allows the user to easily explore the profile data and get different views on it by simply clicking on items within the bargraphs. This invokes more detailed displays:

- The *node profile* shows the function profile for a specific node in more detail.
- The *text node profile* shows the same information in text form similar to the normal UNIX *prof* output.
- The *function profile* can be used to see how a specific function executed on all nodes in more detail.

For easy identification, each function is shown in a unique color in all displays.

The *collection access data profile* summary shows the user an overview of access information to collections (pC++'s distributed data objects) of the current application. For each collection declared in the program, a single bargraph line shows the percentage of all accesses to this collection which were *local* (accessing local node memory) and *remote* (access involved costly communication). In clicking on the collection name at the left side of the bar, the user can get a more detailed view of the profile data for that collection, showing the local/remote access ratios for each node the program was running on.

Figure 4 shows a node profile for node 5, a text node profile for node 1, a function profile for `DistBlkMatrix::setplusvalue`, and a profile for collection `p`.

5.3 Event Tracing of pC++ Programs

In addition to profiling, we have implemented an extensive system for tracing pC++ program events. Currently, tracing pC++ programs is restricted to shared-memory computers (e.g., Sequent Symmetry, BBN Butterfly, and Kendall Square KSR-1) and the uniprocessor UNIX version. The implementation of the event tracing package to distributed memory machines is underway[3]. Trace instrumentation support is similar to profiling. In addition, we have implemented several utilities for event trace analysis (see also Figure 1):

Merging: Traced pC++ programs produce an event log for each node. The trace files will have names of the form `<MachineId>.<NodeId>.trc`. The single node traces must be merged into one global event trace, with all event records sorted by increasing timestamps. This is done with the tool *se_merge*. If the target machine does not have a hardware global clock, *se_merge* will establish a global time reference for the event traces by correcting timestamps (using synchronization and message passing events of the pC++ runtime system).

Trace Conversion: The utility tool *se_convert* converts traces to the SDDF format used with the *Pablo* performance analysis environment [16, 17] or to the ALOG format used in

[3] Note, the difference between the shared- and distributed-memory implementations is only in the low-level trace data collection library and timestamp generation; all trace instrumentation is the same.

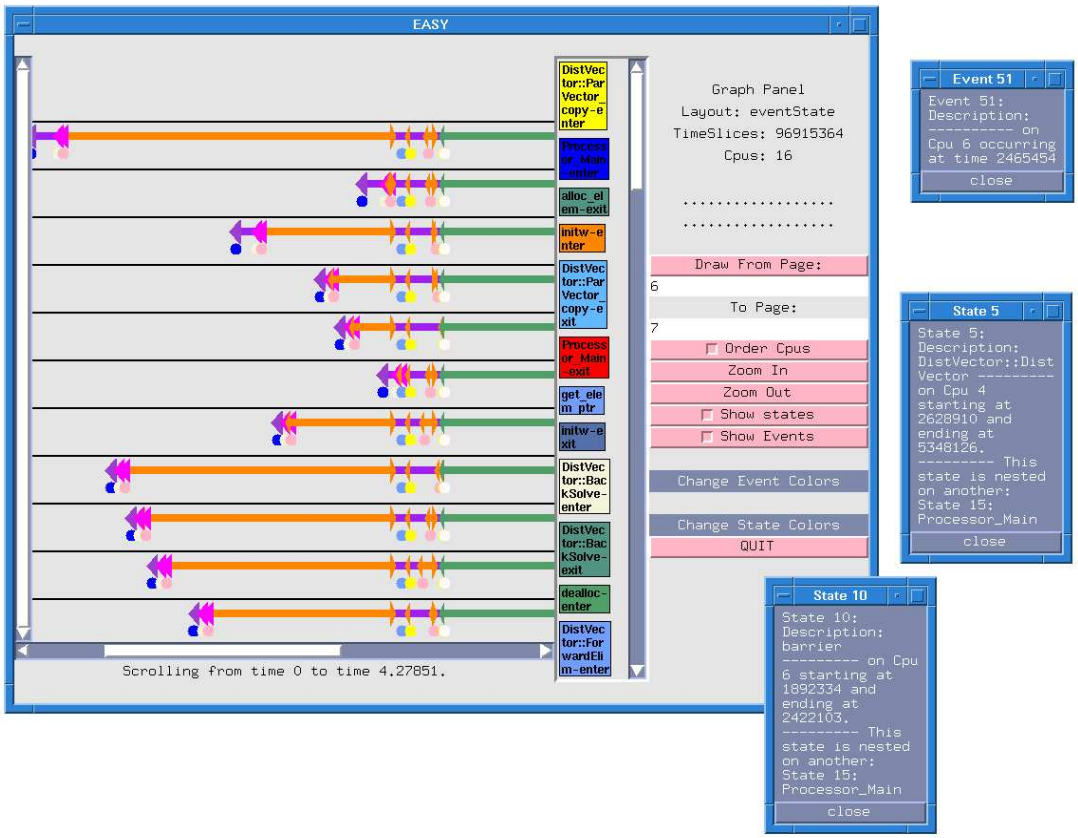

Figure 5: Easy

the *Upshot* event display tool [4]. It also can produce a simple user-readable ASCII dump of the binary trace.

Trace Analysis and Visualization: The trace files can also be processed directly with the *SIMPLE* event trace analysis and visualization environment or other tools based on the *TDL/POET* event trace interface [10, 11]. These tools use the Trace Description Language (TDL) output of the Instrumentor to access the trace files.

5.4 Easy

Easy (**E**vent **A**nd **S**tate displa**Y**)[4] is an Upshot-like event and state display tool based on the ALOG event trace format. *Easy* displays states and events as graphical objects on an X-Y graph. On the Y axis, individual processors are shown, while the X axis extends in time. Both directions are scrollable. A particular event or state can be detailed by clicking on the corresponding graphical object. A window then pops up displaying relevant information such as the time the event occurred, the type of the event and other parameters stored in

[4] Easy is available separately from $\mathcal{T}$. In this form, its moniker is offShoot.

the trace. Also, the states are displayed in such a way that they show when nesting occurs. Figure 5 shows the startup phase of a pC++ program.

6 Future Work

The pC++ project is an on-going research effort. We are planning to enhance the current version into several ways. First, the pC++ language will be extended to include task-parallel programming and more aggressive compiler optimizations. Second, we are working on better tools for the programming environment (make-and-compile manager, graphical instrumentation control, data distribution visualization, barrier breakpoint debugger, etc.) that will be integrated with the current version of $\mathcal{T}$. Lastly, we will try to provide interaction with other high-performance languages like HPF, C++, Fortran-M, etc.

7 Conclusion

The pC++ programming system includes an integrated set of performance instrumentation, measurement, and analysis tools. With this support, we have been able to validate performance scalability claims of the language and characterize important performance factors of the runtime system ports during pC++ system development. As a consequence, the first version of the compiler is being introduced with an extensive set of performance experiments already documented.

The $\mathcal{T}$ environment demonstrates the advantages of language-specific program analysis tools. Its design and integration with the pC++ language system allows important programming productivity issues to be addressed. In addition, $\mathcal{T}$ enables more sophisticated debugging and performance analysis tools to be developed and effectively applied.

For more information ...

Documentation and source code for pC++ and Sage++ are available via anonymous FTP from `moose.cs.indiana.edu` and `ftp.cica.indiana.edu` in the directory `~ftp/pub/sage`. $\mathcal{T}$ is part of the pC++ distribution. We maintain two mailing lists for pC++ and Sage++. For more information or how to join one, send mail to `sage-request@cica.indiana.edu`. No subject or body is required. Also, the pC++/Sage++ project has created a World-Wide-Web server which can be found at `http://www.cica.indiana.edu/sage/home-page.html`. Any WWW viewer can be used to browse the on-line user's guides and papers, get programs, and even view pictures of the development team.

References

[1] J. Hollingsworth and B. Miller. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. *Proceedings of the 1993 International Conference on Supercomputing*, July 1993.

[2] P. Messina and T. Sterling, editors. *System Software and Tools for High Performance Computing Environments.* Society for Industrial and Applied Mathematics, April 1993. from the Pasadena Workshop on System Software and Tools for High Performance Computing Environment, Pasadena, California.

[3] D. Gannon, F. Bodin, S. Srinivas, N. Sundaresan, S. Narayana, *Sage++, An Object Oriented Toolkit for Program Transformations*, Technical Report, Dept. of Computer Science, Indiana University. 1993.

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

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

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

[7] J. K. Lee, *Object Oriented Parallel Programming Paradigms and Environments For Supercomputers*, Ph.D. Thesis, Indiana University, Bloomington, Indiana, Jun 1992.

[8] D. Gannon, J. K. Lee, On Using Object Oriented Parallel Programming to Build Distributed Algebraic Abstractions, *Proc. CONPAR 92–VAPP*, Lyon, France, Sept. 1992.

[9] D. Gannon, *Libraries and Tools for Object Parallel Programming*, 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. 231–246, 1993.

[10] B. Mohr, *Performance Evaluation of Parallel Programs in Parallel and Distributed Systems*, H. Burkhart (Eds.), Proc. CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing, Zurich, Lecture Notes in Computer Science 457, pp. 176–187, Berlin, Heidelberg, New York, London, Paris, Tokio, Springer Verlag, 1990.

[11] 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.

[12] J. K. Ousterhout, *Tcl: An Embeddable Command Language*, Proc. 1990 Winter USENIX Conference.

[13] J. K. Ousterhout, *An X11 Toolkit Based on the Tcl Language*, Proc. 1991 Winter USENIX Conference.

[14] 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.

[15] 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, Apr. 1994.

[16] 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.

[17] D. A. Reed, R. A. Aydt, T. M. Madhyastha, Roger J. Noe, Keith A. Shields, B. W. Schwartz, *An Overview of the Pablo Performance Analysis Environment.* Department of Computer Science, University of Illinois, November 1992.

[18] V. S. Sunderam, *PVM: A Framework for Parallel Distributed Computing*, Concurrency: Practice & Experience, Vol. 2, No. 4, pp. 315–339, December 1990.