Chapter 1

Tau

October 27, 1995

Tuning and Analysis Utilities for Portable Parallel Programming

Bernd Mohr, Allen D. Malony, Janice E. Cuny
{mohr,malony,cuny}@cs.uoregon.edu

☆ Tuning and Analysis Utilities ☆

http://www.cs.uoregon.edu/paracomp/tau

This research is supported by ARPA under Rome Labs contract AF 30602-92-C-0135 and Fort Huachuca contract ARMY DABT63-94-C-0029, and in part by the National Science Foundation under grant NSF ASC92-13500, by a grant from OACIS, and by IBM R&D contract MHVU3704.

1.1 Introduction

Most users find parallel programming difficult for at least four reasons. First, parallel computing abstractions (e.g., data parallelism, control or task parallelism, producer/consumer parallelism) are diverse, differing mainly by the type of parallel behavior supported (or allowed) in a program's execution. In addition to learning the parallel programming languages and tools in a particular environment, a user must decide which parallel computing model provides the "best" execution for their problem. Deciding how to choose between models often requires a sophisticated understanding of the application, its underlying algorithms, the expressiveness of the language used, and effects of the system software and hardware architecture of the parallel machine. Unfortunately, this choice is complicated by the fact that not all parallel computing abstractions are equally well supported in existing programming systems. Second, most parallel programming systems do not insulate users fully from low-level hardware and system software concerns; those that do make it difficult for users to undertake performance debugging, and hence to realize the potential high performance of parallel systems. Third, program analysis tools for parallel programming are either not generally available, not particularly useful, or not integrated into a complete programming environment. Fourth, users' requirements for parallel computing are constantly changing: high performance is important, but so is the need to have parallel programs interoperate with graphics systems, networked resource servers, databases, and so on.

The most common way to address these issues at present is to develop languages which support specific parallel computing abstractions, such as HPF. These languages invoke parallel operation via compiler transformations and calls to runtime system routines. Users gain programmability, since they specify parallelism abstractly and rely on the compiler to generate task and data mapping code, and portability, since the system can be retargeted to new platforms. However, these gains often come at the expense of observability. Unless the programming system contains analysis tools that can relate a program's execution dynamics to its semantics, increased abstraction will make it difficult to debug or tune program performance. This suggests that tools should be designed specifically to meet the requirements of the language environment.

Since 1992, we have participated in building such an integrated toolset for pC++, a parallel C++-based language (Chapter ??). Our charter was to design and develop a program analysis tool system. The project was unique in that we decided early on to let the programming language requirements determine the tool specifications. This is in contrast to other projects to build program analysis tools, principally performance tools, where general measurement and analysis support is targeted, but this support may be deficient or cumbersome to use in specific language contexts. In addition, we leveraged the language system infrastructure to enhance the integration of compiler and analysis tools. The fact that our base language was C++ encouraged us to make our tools as flexible as possible, so that we could use encapsulation (for supporting data aggregation and parallel invocation) and inheritance (for building distributed data structures) in our implementations. Finally, since pC++ was designed to run on all parallel MIMD systems, our program analysis environment had to be portable as well.

The result of our efforts is called $\mathcal{T}$ (TAU), for **T**uning and **A**nalysis **U**tilities. In Section 1.2, we list the requirements that we felt $\mathcal{T}$ had to address. How $\mathcal{T}$'s design meets these requirements is discussed in Section 1.3, which also covers $\mathcal{T}$'s architecture and implementation. Each $\mathcal{T}$ tool is described in full in Section 1.4. Since the effectiveness of $\mathcal{T}$ can be only measured by its usefulness for analysis of pC++ programs, Section 1.5 shows how $\mathcal{T}$ was used to analyze the pC++ implementation of polygon overlay, described in Section **??**.

$\mathcal{T}$ is not without its shortcomings. A critique of its current state is given in Section 1.6. Section 1.7 discusses future development of $\mathcal{T}$, particularly its extension to other parallel language environments and the incorporation of more sophisticated tools.

1.2 Design Requirements and Goals

An earlier version of this material was published in [9].

$\mathcal{T}$ was designed to improve parallel programming productivity by combining advances in parallel debugging, performance evaluation, and program visualization tools. We feel that the requirements the design and the tools had to address are common to next-generation parallel programming environments, and include:

Give a user (program-level) view. Past tool development has been dominated by efforts directed at the execution level (e.g., efficient implementation of monitoring). Consequently, tool users are given little support for translating between program-level semantics and low-level execution measurements.

Support high-level, parallel programming languages. The development of advanced parallel languages (e.g., HPF and pC++) separates users from execution-time reality. A successful tool must present information to users in the terms defined by the language they are using.

Integrate with compilers and runtime systems. Most 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.

Enable portability, extensibility, and retargetability. Users of portable languages need a consistent program development and analysis environment across multiple execution platforms. Tools should be extensible, so that they can accommodate new language or runtime system features, and retargetable, so that the tool design can be reused for different languages and environments.

Enhance usability. A high-level, portable, integrated tool is not automatically easy to use. In the past, too little emphasis has been put on interface design, which has led to tools being poorly used.

These requirements become even more significant as we develop parallel languages with highly optimized runtime systems. We believe that the main problems are ones of tool design rather than functionality: existing tools implement a broad range of analysis techniques, but have not been successfully integrated into usable parallel programming environments. One way to improve integration is to base the design of tools on the particular performance and debugging requirements of the parallel language for which the tools will be used. In this manner, tools can specifically target program analysis support where tool application is well understood. However, this cannot be fully realized unless tools can leverage other programming system technologies (e.g., use the compiler to implement instrumentation).

1.3 TAU Overview

> Parts of this material (Section 1.3 and Section 1.3.1) were previously published in [9].

The $\mathcal{T}$ architecture defines how its components interoperate and fit in the pC++ language system. Below, we describe the $\mathcal{T}$ design and show how it addresses the programming productivity requirements of pC++. $\mathcal{T}$ is not a general solution to the problem of parallel program analysis. Instead, our goal was 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.

$\mathcal{T}$ was specifically designed to meet the requirements listed in the previous section:

Give 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 high-level, parallel programming languages. $\mathcal{T}$ is defined by the program analysis requirements of pC++, and was designed and implemented in concert with the pC++ language system. The most difficult challenges during the development of $\mathcal{T}$ were to determine what low-level instrumentation was needed to capture high-level execution abstractions, and how to translate performance data back to the application and language level.

Integrate with compilers and runtime systems. $\mathcal{T}$ uses the SAGE++ toolkit [3] as an interface to the pC++ compiler for instrumentation and accessing properties of program objects. $\mathcal{T}$ is also integrated with the runtime system of pC++ for profiling and tracing support.

Enable portability, extensibility, and retargetability. We implemented $\mathcal{T}$ in C++ and C to ensure an efficient, portable, and reusable implementation. The same reasoning led us to use Tcl/Tk [11] for our graphical interface.

The $\mathcal{T}$ tools are implemented as graphical *hypertools*. While each is distinct, they act in concert like a single application. Each tool implements some defined tasks; if one tool needs a feature of another, the first sends a request to the second (e.g., "display the source code for a specific function"). This design allows easy extension. The SAGE++ toolkit also supports Fortran-based languages, so that $\mathcal{T}$ can be retargeted to other programming environments.

Enhance 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, in that clicking on them brings up windows which describe the element in more detail. This allows the user to explore properties of the application by interacting with the elements of most interest. The $\mathcal{T}$ tools also support *global features*. When a global feature is invoked in any tool, it is automatically executed in all $\mathcal{T}$ tools which are currently running. Examples of these global features are described later. $\mathcal{T}$ also includes a full hypertext help system.

1.3.1 TAU Architecture

Figure 1.1 shows an overview of the pC++ programming environment. The pC++ compiler frontend takes a user program and pC++ class library definitions (providing predefined collection types) and parses them to create a program data base, or PDB. The PDB is accessed via the SAGE++ library.

Through command line switches, the user can choose to compile a pC++ program for profiling or tracing. In either case, the instrumentor is invoked to add the necessary instrumentation to the PDB (Section 1.4.2). The pC++ backend transforms the PDB 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 user-friendly and convenient way of compiling and linking pC++ programs (Figure 1.7[1]). Through a graphical interface, the user first selects the parallel machine on which the application is to be compiled and run. Parameters and options for compiling and running are chosen through pull-down menus. COSY automatically connects, if necessary, to the remote machine, executes the appropriate commands, and displays the resulting output in a scrollable window.

The program and performance analysis environment is shown in the bottom right corner of Figure 1.1. It includes the $\mathcal{T}$ toolset, instrumentation, profiling, tracing, and breakpointing support, and interfaces to performance analysis tools developed by other groups [5, 8, 2, 12]. The $\mathcal{T}$ tools are described in more detail in Section 1.4.

1.3.2 TAU Implementation

The $\mathcal{T}$ architecture defines how the tools interoperate and fit into the pC++ programming system. This section discusses two components of $\mathcal{T}$ that support this: global features and well-defined internal tool interfaces.

Global Features

Global features are a natural extension to the "click-for-source" feature found in some other parallel program analysis tools. But whereas these tools only allow users to find the source code of elements in the performance views (e.g., show the code containing the `send` or `receive` function call for a message transmission), $\mathcal{T}$ allows users to click on anything which represents

[1]Figures 1.6 to 1.22 appear in Section 1.5

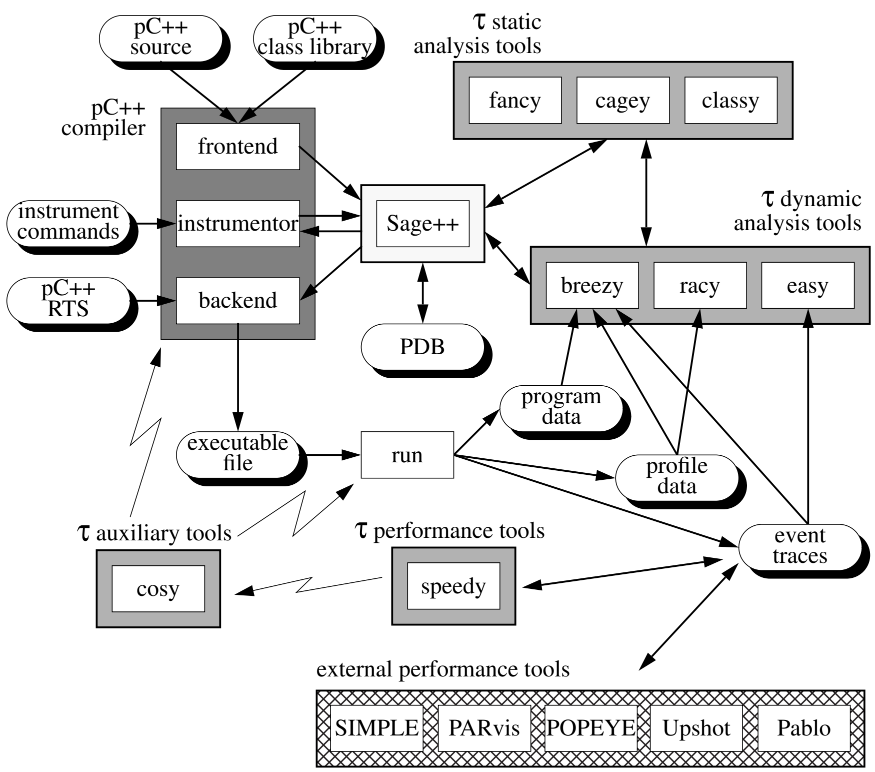


Figure 1.1: TAU Tools Architecture

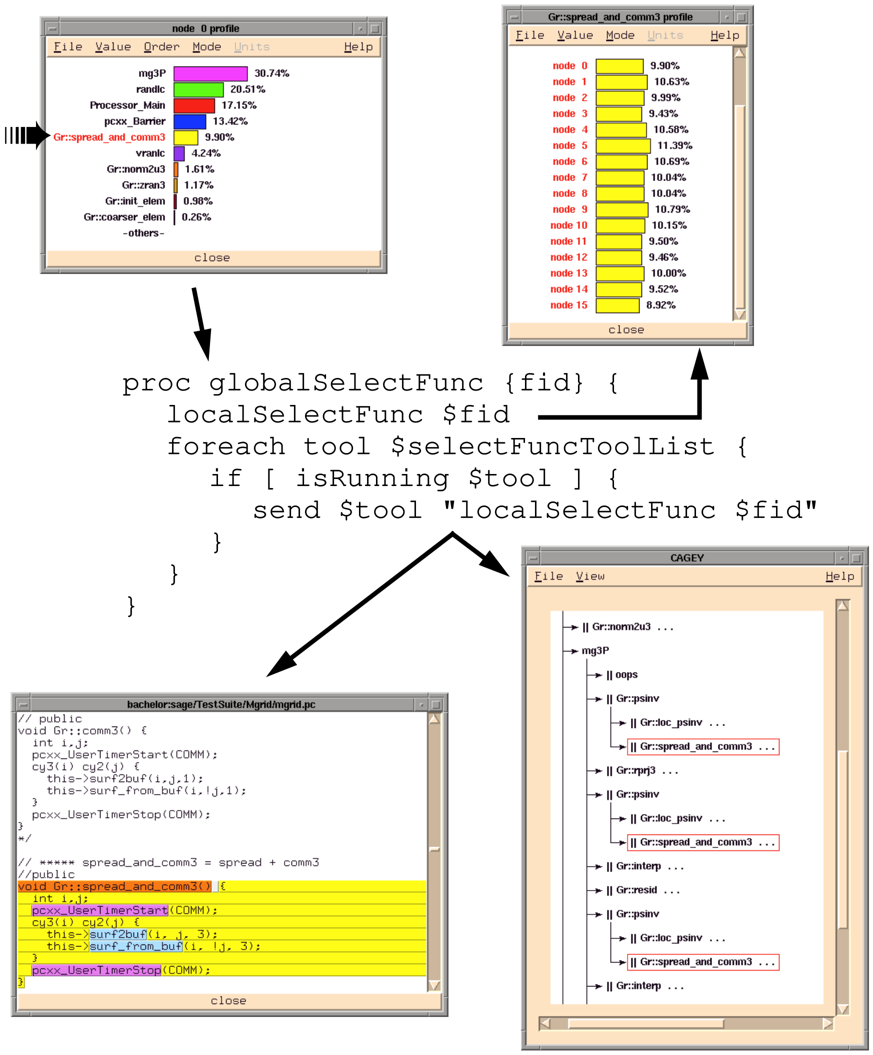


Figure 1.2: Global feature Implementation

a function or class in any tool, and automatically updates all tool displays to show information about the selected object (Figure 1.2).

For example, suppose a user is looking at the execution profile for her application, and wonders why a specific function is taking so much time. Clicking either on the function's name label or on the colored bar showing the execution time used by the function will invoke the global feature `select-function`. This calls the Tcl/Tk function `globalSelectFunc`, shown in the middle of Figure 1.2, with the unique identifier of the selected function as a parameter. This then calls `localSelectFunc`, which causes the tool to show more specific information about the selected function. Next, each of the tools known to implement the global feature is checked to see if it is running. If it is, a message is sent to invoke `localSelectFunc` with the same function name as an argument. In Figure 1.2, this causes the file browser to show the source code of the selected function, and the callgraph browser to show all of its call sites. The same effect would have been achieved if the user had clicked on a function in either the source code browser or the callgraph display.

This implementation has several strengths:

- The use of global features makes it possible to implement the $\mathcal{T}$ environment as *hypertools*, instead of as a single huge program. Hence, the individual tools can be kept small, which simplifies maintenance and debugging. It also makes tools easier to re-use in different contexts or environments.
- The environment can easily be extended. A new tool only has to implement those global features it needs or is able to support, then bind invocation of those features to appropriate elements of its graphical user interface and add itself to the global list of tools supporting that feature.
- The use of high-level interprocess communication (e.g., Tk's `send`) allows a very simple implementation. It is also quite portable, as there are now Tk modules for Scheme and Perl, and a C interface for writing libraries.

Other examples of global features include `select-class`, `switch-application`, and `exit`.

Internal Tool Interfaces

Figure 1.3 shows the internal implementation of the $\mathcal{T}$'s static analysis tools: the source code browser Fancy, the callgraph browser Cagey, and the class hierarchy browser Classy (not shown in Figure 1.3). If a browser is started or switched to another user application, it invokes the object manager (`om`) and reads its output. Command line switches (e.g., `-txt` or `-cg`) are used to specify the type of the requested information. The object manager uses the Sage++ interface to access the program database describing the current user application, then prints the requested information

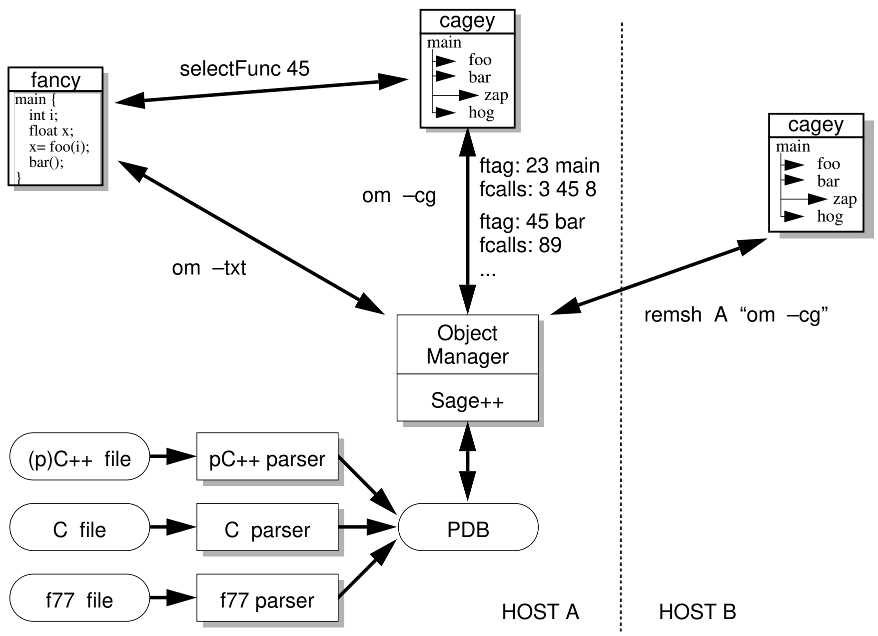


Figure 1.3: TAU Internal Tool Interfaces

in an easy-to-parse ASCII format. The object manager reduces dependence on a particular language system by producing the necessary information in a generic format where possible, and enhances tool interoperability by making inter-tool communication more robust. For example, this architecture of information servers makes it easy to use $\mathcal{T}$ on a workstation to control a pC++ application running on remote parallel computer. In this case, instead of launching the object manager directly, the $\mathcal{T}$ tools use a standard TCP/IP remote shell command.

$\mathcal{T}$'s dynamic analysis tools are implemented in much the same way. Racy and Easy invoke programs to read profile data and event traces respectively, then parse their ASCII output. This allows $\mathcal{T}$ to be ported to other language systems in any of four different ways:

1. Change the compiler of the new target environment to produce a program data base in the format used by Sage++.

2. Implement that part of the Sage++ function interface used by the object manager for accessing the program data base used in the new language environment.

3. Implement new information servers that understand the same command line options, and output information in the same format, as the $\mathcal{T}$ object manager.

4. Change the $\mathcal{T}$ tool interfaces to the information servers so that they it work with the new language environment.

As an example, $\mathcal{T}$ was ported recently to work with the HPF compiler of the Portland Group Inc. We used the third approach—write a new object manager—to port the source code and the callgraph browser, and the first approach—implement routines to generate $\mathcal{T}$-compatible profile data files—to adapt the profile data browser. The whole port required less than one person-week, and shows the benefits of $\mathcal{T}$'s modular design.

1.4 TAU Tools

In this section, we describe the tools in the current $\mathcal{T}$ toolset[2]. These tools are available as part of the pC++ distribution and operate in any environment where pC++ runs. Some tools are pC++ specific, while others are could be applied to other language systems. As $\mathcal{T}$ was designed to support extensions to the toolset, new tools are continually being developed and incorporated into the $\mathcal{T}$ architecture.

1.4.1 Static Analysis Tools

> An earlier version of this material was previously published in [9]. This material has been updated where tool features have changed or new tool features added.

A major motivation for using C++ as the base for new parallel languages is its ability to support develoment and maintenance of large, complex applications. However, if they are to use C++'s capabilities effectively, users must be given support tools which can access source code at the level of programming abstractions.

$\mathcal{T}$ currently provides three tools to enable users to navigate through large pC++ programs: a global function and method browser called Fancy, a static callgraph display called Cagey, and a class hierarchy display called Classy. Since these tools are integrated with $\mathcal{T}$'s dynamic analysis tools, it is easy for users to find object-level execution information. To locate dynamic results after a measurement has been made, a user only has to

[2] All $\mathcal{T}$ tools have adjectives as names, so that the answer to "What is $\mathcal{T}$?" is "$\mathcal{T}$ is Cosy, Fancy, Cagey, Classy, Easy, Racy, Speedy, and Breezy!"

click on the object of interest, such as a function name in a callgraph display.

File and Source Code Browser

FANCY (**F**ile **AN**d **C**lass displa**Y**) lets a user browse through the files and classes making up her application. The main window displays listboxes showing the source files used and the classes defined (Figure 1.8). Selecting an item in either listbox displays all global functions defined for the selected file, or all methods of the selected class.

Selecting a global function or a class method causes the corresponding source code to be displayed in a separate viewer window (Figure 1.9). The header and body of the currently selected routine, as well as functions and methods which are called from that routine, are highlighted using different colors. Routines and class definitions can be selected by clicking on the appropriate names.

Callgraph Browser

CAGEY (**CA**ll **G**raph **E**xtended displa**Y**) shows the static callgraph of the user's application (Figure 1.10). It uses SAGE++ to determine the callgraph structure and to differentiate between global functions and class methods. CAGEY helps users locate parts of their programs where parallelism is used by marking parallel routines with the string "||". As callgraphs can be quite large, CAGEY allows users to control how far a callgraph is expanded.

CAGEY supports two graph layout modes: *extended* and *compact*. In compact mode, each function or method is represented by a single node. If a function calls another more than once, the connecting arc is labeled with the number of calls. This mode works well for structured or regular codes. In expanded mode, CAGEY draws a node for each individual function or method call; the resulting graph is always a tree. Figure 1.10 shows a callgraph in compact mode.

Class Hierarchy Browser

CLASSY (**CLASS** hierarch**Y** browser) is a class hierarchy browser for programs written in C++ and derivatige languages such as pC++. Classes which have no base class (called level 0 classes) are shown on the left side of the display window (Figure 1.11). Subclasses derived from level 0 classes are shown in the next column to the right, and so on. Like CAGEY, CLASSY lets the user choose the level of detail in the class hierarchy display by allowing folding or expansion of subtrees in the graph. CLASSY also allows quick access to key properties of a class, such as data members. Finally, CLASSY marks collections by putting the string "||" before their names.

1.4.2 Dynamic Analysis Tools

An earlier version of this material was published in [9].

Static analysis tools provide high-level views of a program's structure; dynamic analysis tools capture information about the program's execution and correlate it with those high-level views so that users can find correctness and performance problems. $\mathcal{T}$ supports dynamic analysis in three ways: *profiling*, which computes statistical information to summarize program behavior; *tracing*, which portrays execution behavior as a sequence of abstract *events* that can be used to determine various properties of time-based behavior; and *breakpoint debugging* which allows a user to stop the program at selected points and query the program's state. These are supported by an execution profile data browser called RACY, an event trace browser called EASY, and a barrier breakpoint debugger called BREEZY.

Program Instrumentation

An earlier version of this section was published in [7].

All three analysis modes use instrumentation to capture runtime data. The program transformations needed for this are done at the language level to ensure portability. One problem this approach faces is to ensure that code to profile function exits is executed as late as possible. Since a function can return an arbitrarily-complex expression, correct profiling instrumentation must somehow extract the expression from the return statement, compute its value, execute the profiling exit code, and only then return the expression result. Matters become even more complicated when we consider multiple exit points.

This is a good example of how we can leverage our language environment for tool implementation. The trick is very simple: we declare a special *Profiler* class that has a constructor and a destructor, but no other methods. A variable of this class is then declared in the first line of each function that has to be profiled, as shown below for function `bar`.

```
class Profiler {
 char* name;
 public:
 Profiler(char *n) {name=n; code_enter(n);}
 ~Profiler() {code_exit(name);}
};
```

```
void bar() {
  Profiler tr("bar"); // Profiler variable
  // body of bar
}
```

The variable `tr` is created and initialized by its constructor each time control flow reaches its definition, and destroyed by its destructor on exit from its block. The C++ compiler automatically rearranges the code and inserts destructor calls to ensure correct behavior no matter how the scope is exited.

We use the SAGE++ toolkit to manipulate pC++ programs to insert such instrumentation at the beginning of each function. The user can selectively specify the set of functions to instrument using an instrumentation command file. Filenames, classes, and functions can be specified as regular expressions, and included or excluded based on their name, source file, or class. Functions can also be selected by their size (measured in number of statements and/or number of function calls), by whether they are inline functions, and by their position in the static callgraph. If an instrumentation command file is not given, every function in the pC++ input files is profiled by default. A graphical interface allowing to control the instrumentation process for $\mathcal{T}$ is not yet available.

Portable Profiling for pC++

The data captured by the entry and exit instrumentation described above can be used to calculate the number of times a function is called and the execution time it consumes. For pC++ we capture performance profiles for thread-level functions, collection class methods, and runtime system routines. The data we capture includes activation counts, execution times, and, in the case of collections, referencing information.

Our approach to profiling has two basic advantages. First, instrumenting at the source code level makes it very portable. Second, different implementations of the profiler can be easily created by providing different code for the constructor and destructor of the `Profiler` class. This makes instrumentation very flexible. Currently, we have implemented two versions of the profiler: one based on direct profiling, and one which calls event logging functions from the pC++ library. Other profiling alternatives could be implemented in the same way. Instrumented version of the pC++ class libraries support profiling of runtime system functions and collection access.

If a pC++ program was compiled for direct profiling, executing it produces a profile data file for each node. This profile data can be browsed using either PPROF, a parallel profile tool similar to UNIX PROF, or RACY

(**R**outine and data **AC**cess profile displa**Y**). RACY, shown in Figure 1.12, gives a quick overview of an applications' execution by summarizing both function and collection access performance. The function profile summary presents a bargraph with one line per processor, showing where program time was spent on that processor. In addition, the mean, maximum, and minimum values are shown on top of the graph. Detailed profiles for each node and each function can be displayed in a variety of formats (Figures 1.13, 1.15, and 1.16).

The collection access data profile summary shows access information for pC++ collections. A bar graph shows the percentage of collection accesses that were local or remote. By clicking on the collection name, the user can get a per-node profile of this data. (Figure 1.17).

Event Tracing for pC++

Some of this material was previously published in [7].

In addition to profiling, we have implemented a system for tracing pC++ program events. Events are stored in a buffer on each node, which is written to disk when it is full or when the program ends. Each event record includes the event type, the originating processor, a high-resolution timestamp, and an optional parameter. Each event is assigned an *event class* (see Table 1.1); when a program compiled for tracing is executed, particular event classes can be activated or deactivated to allow selective recording. The instrumentation required by tracing is implemented in the same way as profiling instrumentation.

We have also implemented several utilities for merging event traces, for converting them to other formats, such as PARVIS's PV [2], PABLO's SDDF [12], or UPSHOT's ALOG [5], and for analyzing and visualizing traces using the SIMPLE environment or other tools based on the TDL/POET interface [8].

EASY (**E**vent **A**nd **S**tate displa**Y**) is an UPSHOT-like event and state display tool. It displays states and events on an X-Y graph, allowing more detailed access to event data when necessary. The Y axis shows individual processors, while the X axis shows elapsed time. A particular event or state can be examined by clicking on the corresponding graphical object. States are displayed in such a way that they show when states are nested. Figure 1.18 shows the major phases of a pC++ program.

Barrier Breakpoint Debugging

An extended version of this section was previously published in [4].

`EC_BASIC`	Basic runtime events like begin and end of the whole program and the user's main function.
`EC_KERNEL`	Creation and deletion of collections, collection element access.
`EC_RUNTIME`	Entry and exit of every pC++ runtime function including barriers, message send and receives, polling. Mainly used for debugging.
`EC_TIMER`	Calls to the pC++ timer and clock package.
`EC_TRACER`	Functions of the software event tracing package itself like Init, Close, and FlushBuffer.
`EC_PROFILER`	User function entry and exit points. Events of this class are automatically inserted by the pC++ instrumentor.
`EC_USER1` ··· `EC_USER4`	Available to the user for manually inserted event recording calls.

Table 1.1: pC++ Event Classes

We have developed a program interaction system for pC++ called BREEZY (**BR**eakpoint **E**xecutive **E**nvironment for visuali**Z**ation and data **D**ispla**Y**) [4]. BREEZY provides the infrastructure for a client application to attach to a pC++ application at runtime. This partnership gives the client several capabilities:

- The client can control the execution of the program.
- The client can retrieve data from parallel data structures created in the program.
- The client can invoke functions or methods in the parallel program.
- The client can retrieve information about the program's execution state.
- The client can retrieve meta-information about the program, such as type descriptions.
- The client may communicate in a general way with the parallel program.

The BREEZY architecture consists of three modules (Figure 1.4). The *Breakpoint Executive* maintains information about the program's state, including a list of currently-instantiated parallel data objects. It consults the

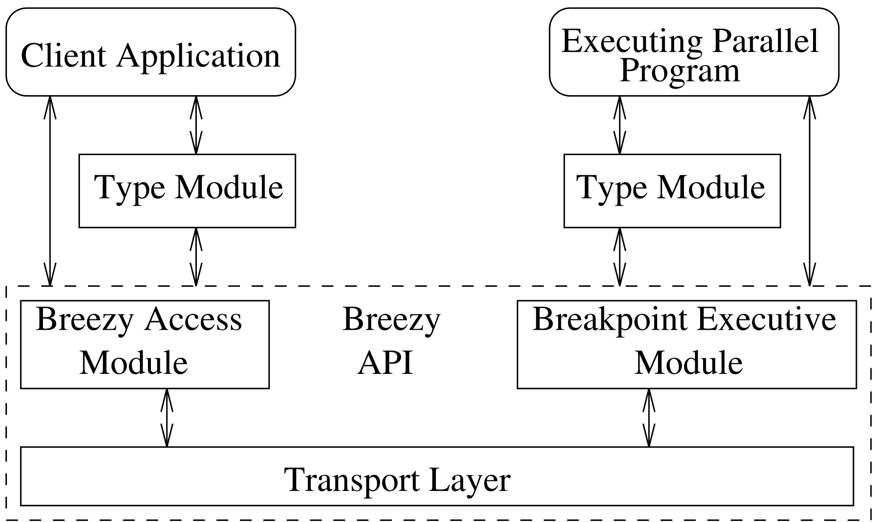


Figure 1.4: Breezy Architecture

Type Module, which stores meta-information such as type descriptions of the parallel data structures or lists of all user-defined functions that can be called.

The *Breezy Access Module* is currently implemented as a library of C routines. It allows a client to control the execution of the program, to request information about the program state, and to access program data structures. For example, a client using the API can specify the program variable that holds the parallel data object of interest. If this object is a structured object with fields, such as a class, the client can further specify a particular field. The client can then retrieve this data from all of the distributed elements of the parallel data object, or from a single element in that object. To serve requests for parallel data, the *Breakpoint Executive* calls access functions in the executing program. These access functions reside in the (modified) user program in order to have access to the program variables and functions, and are generated automatically by the Breezy instrumentation phase of the pC++ compiler.

Important features of Breezy are:

- Its modular design allows for component re-use and clean substitution of new technologies (such as replacing the transport layer with CORBA/IDL[10]).

- It allows the programmer to make functions available for calling by the client, giving the client the power to alter the course of the program or perform specific computations. This allows to implement more complex functionality, such as computational steering.

- Almost all of the implementation is done in the target language making it as portable as the language itself (There need to be some minor support from the runtime system).

This last point is particularly interesting because it allows client applications to reference data objects exactly as they were defined in the program, rather than at a lower level resulting from compiler transformation. Also, it means that a new implementation of Breezy is not required for each new architecture; because Breezy is implemented using the language, it runs everywhere the language does.

1.4.3 Performance Extrapolation for pC++

An extended version of this section was previously published in [15].

The dynamic analysis tools already discussed enable users to investigate the execution behavior of pC++ programs. However, because pC++ programs are portable, users may want to develop and analyze programs that will run across platforms or that will run in environments different from the development environment. To support this type of program analysis, we implemented a performance extrapolation system for pC++ called ExtraP, that has been integrated into $\mathcal{T}$ in the guise of the Speedy tool [15]. The ExtraP/Speedy combination allows users to predict the performance of pC++ programs in target execution environments.

The technique that we developed extrapolates the performance of a n-thread pC++ program from a 1-processor execution to a n-processor execution. The pC++ language system is used to build a n-thread program which is executed on a single processor using a non-preemptive threads package. Important high-level events including remote accesses and barriers are recorded along with timestamps during the program run in a trace file. The instrumented runtime system is configured so that remote accesses are treated as taking place instantaneously, and so that execution threads are released from a barrier as soon as the last thread enters it.

Events are then sorted on a per-thread basis, and their timestamps adjusted to reflect concurrent execution. This is possible because the non-preemptive thread package used only switches threads at synchronization points, and because global barriers are the only form of synchronization used by pC++ programs. Thus, the behavior of threads between barriers is

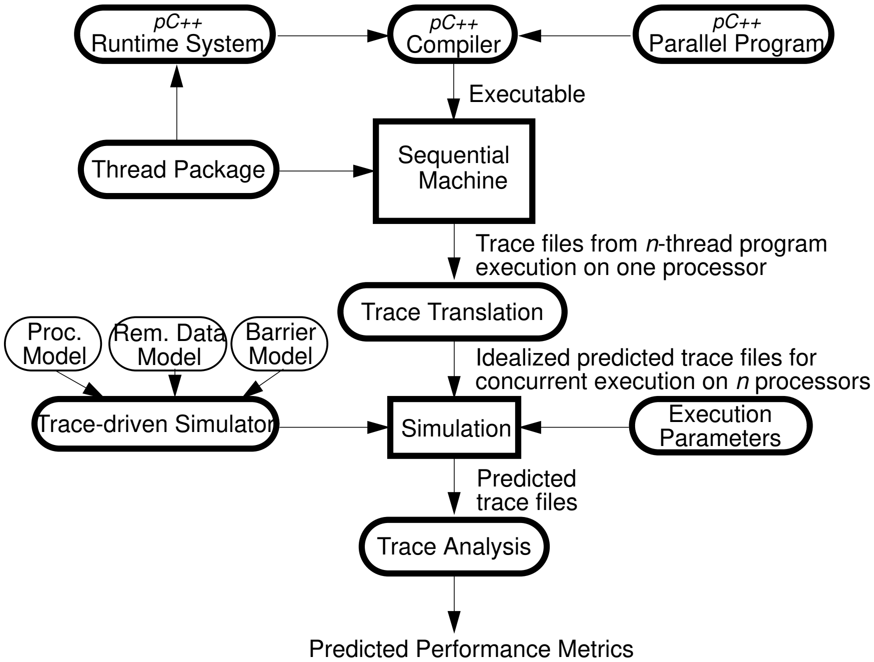


Figure 1.5: A Performance Extrapolation Technique for pC++

independent, and the sorted trace files look as if they were obtained from an n-thread, n-processor run. The only features these traces lack are timings for remote accesses and barriers. A trace-driven simulation attempts to model such features and predict when events would have occurred in a real n-processor execution environment. These extrapolated trace files are then used to obtain performance metrics for the pC++ program. The technique is depicted in Figure 1.5, and described in more detail in [13, 14, 16].

ExtraP uses pC++'s built-in event tracing system to generate the traces needed for the simulation. These traces can be analyzed using $\mathcal{T}$'s event trace browsers, and compared and validated against traces from real parallel executions. Actual extrapolation experiments can be controlled using Speedy (**S**peedup and **P**arallel **E**xecution **E**xtrapolation **D**ispla**Y**), shown in Figure 1.20. Speedy lets users control the compilation of pC++ programs, specify parameters for the extrapolation model and the experiment, execute the experiment, and view the experiment results. Speedy uses Cosy to perform the necessary compilation, execution, trace processing, and extrapolation commands. Speedy also automatically keeps track of parameters by storing them in *experiment description files* and manages all trace and experiment control files. Users can re-execute experiments, or re-use parameter specifications, by loading a former experiment description file into SPEEDY.

The experiment and extrapolation model parameters can be entered and viewed through the EXTRAP parameter file viewer (see Figure 1.21). Numerical parameters can either be entered directly into the input entry or manipulated through a slider bar and increment/decrement buttons. Parameters with discrete values can be specified through a pull-down menu (like `ProcessMsgType` in the picture). In Figure 1.21, the viewer displays the parameters associated with the modeling of the processor of the target machine. Other modeling parameter groups can be displayed by pressing one of the buttons in the top of the viewer window. Besides the five parameter groups describing the simulated parallel machine the group `General` allows the setting of parameters controlling the generation and post-processing of the execution traces.

1.5 Tour de TAU: The Polygon Overlay Example

In this section, we show how the $\mathcal{T}$ environment was used to analyze the pC++ implementation of the polygon overlay problem described in Section ??.

1.5.1 Utility Tools

When $\mathcal{T}$ is started from the command line, the $\mathcal{T}$ main control panel appears (Figure 1.6). The first line shows the host name and architecture of the parallel machine on which compilation and execution of the pC++ program will take place. The second line displays the directory where the program files and program database are stored. The other fields show information about the currently-selected user application. The last line only shows information when a program was compiled for profiling. The buttons at the bottom are used to invoke $\mathcal{T}$'s static and dynamic tools.

The compilation and execution of pC++ programs can be controlled using COSY (Figure 1.7). COSY automatically connects to the remote machine (if necessary), executes the appropriate commands, and displays the resulting output. It allows users to **`build`**, **`run`**, and **`stop`** pC++ programs, to set compilation and execution parameters, and to do standard tasks like cleaning up or listing the current directory.

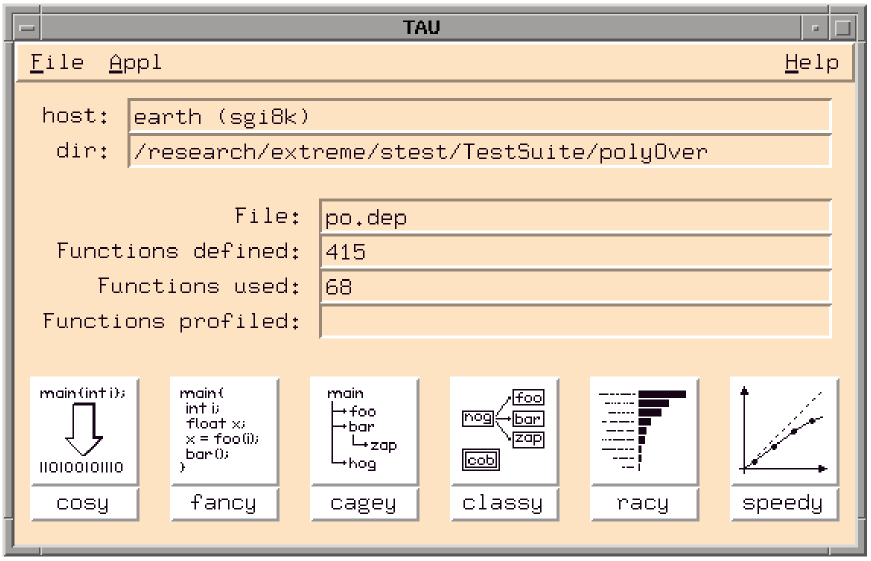


Figure 1.6: TAU Main Control Panel

Figure 1.7 shows the compilation and execution of the polygon overlay example. The version shown is being compiled for tracing; the generated event traces can then be viewed with EASY. As the pC++ polygon overlay program re-uses the original ANSI C reference program, we have to specify a list of extra objects in the `Build Parameters` window before we can `build`. The user must supply a makefile for these objects. We then supply the necessary extra command line parameters in the `Run Parameters` window as well as the number of processors we want to use and the event classes we want to be activated. Pressing the `run` button finally executes the program and, in Figure 1.7, we can see the actual command executed and the output of the program.

1.5.2 Static Analysis Tools

Figure 1.8 shows the main window of the FANCY file and class browser. The lists of all files and classes are shown on the left. The files list includes the pC++ main program (`po.pc`), and pC++-specific header files like `kernel.h`, as well as the files making up the ANSI C reference implementation. The classes list includes pC++'s predefined collection classes (`Kernel` and `SuperKernel`) the pC++-supplied classes for describing the alignment and distribution of collections, and the user-defined collection

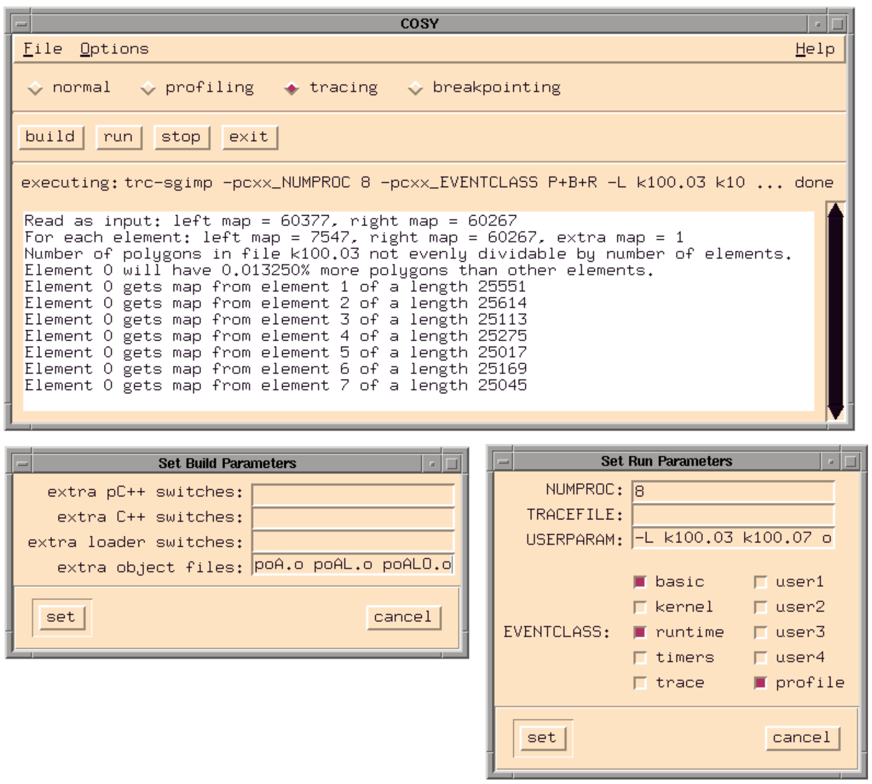


Figure 1.7: COSY

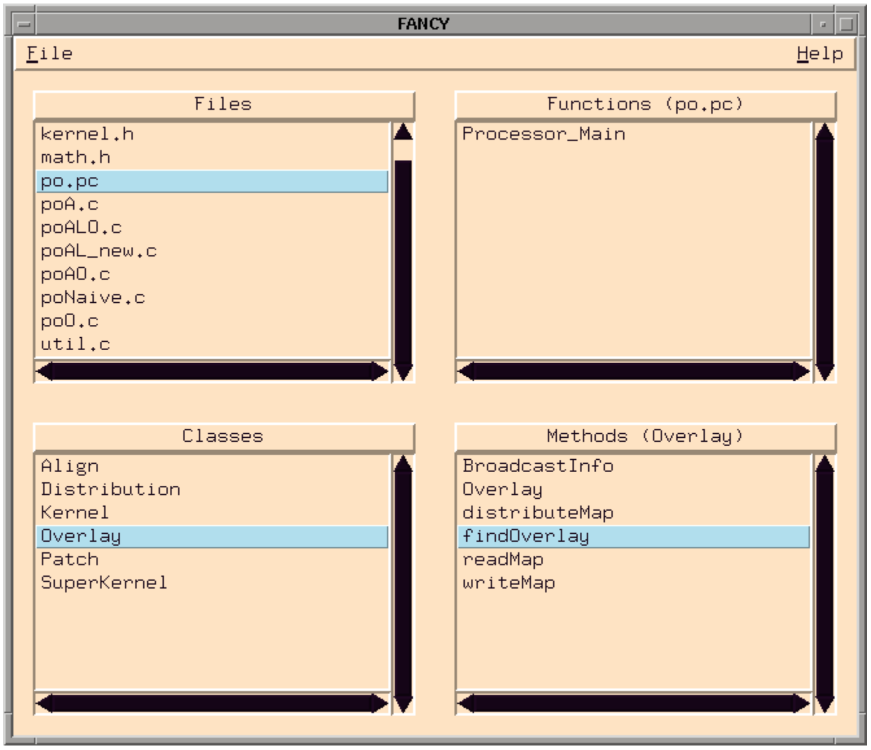


Figure 1.8: FANCY Main Window

(**Overlay**) and element (**Patch**) classes. Selecting a file shows all global functions defined in the file, selecting a class, all its defined methods. Selecting a function or method (like **findOverlay** in the example) shows the source code of this routine in the FANCY file viewer (see Figure 1.9).

Figure 1.10 shows a CAGEY callgraph view of the polygon overlay code with **Overlay::findOverlay** expanded. In addition to allowing users to check the static structure of their programs, these displays are convenient navigation aids. As can be seen, the overlay example has a fairly simple structure, with the bulk of the data-parallel computation in the expanded routine.

The class hierarchy browser, CLASSY, allows quick access to key properties of a class. pC++ collections are marked with a "||" before the name. When a class is selected, a member table window is displayed, which shows a detailed list of the class's members and their attributes (Figure 1.11). The word **element** is used to indicate the pC++ concept of a *method of element*

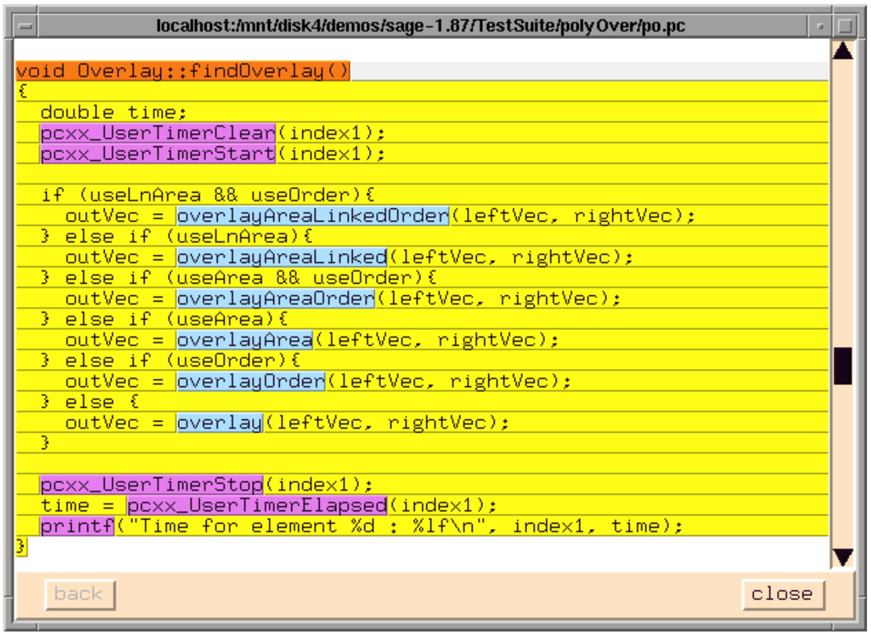


Figure 1.9: Fancy File Viewer Window

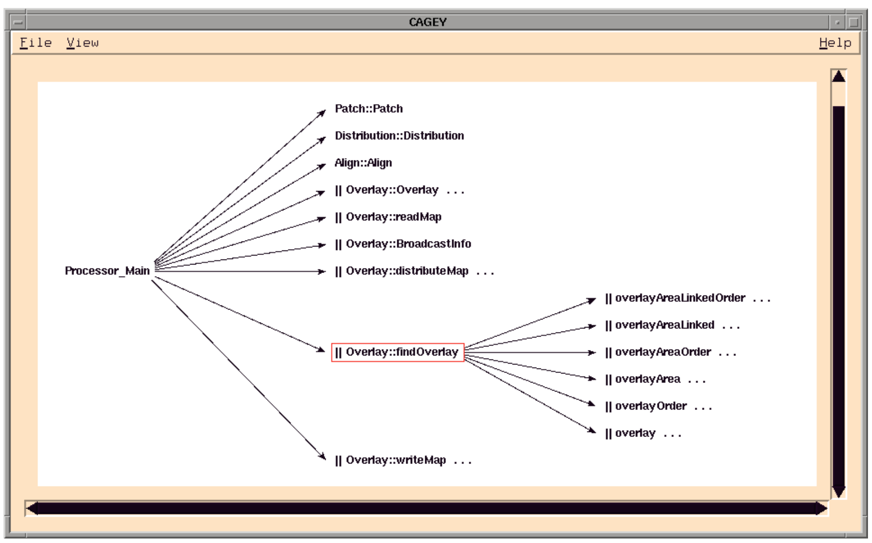


Figure 1.10: Cagey

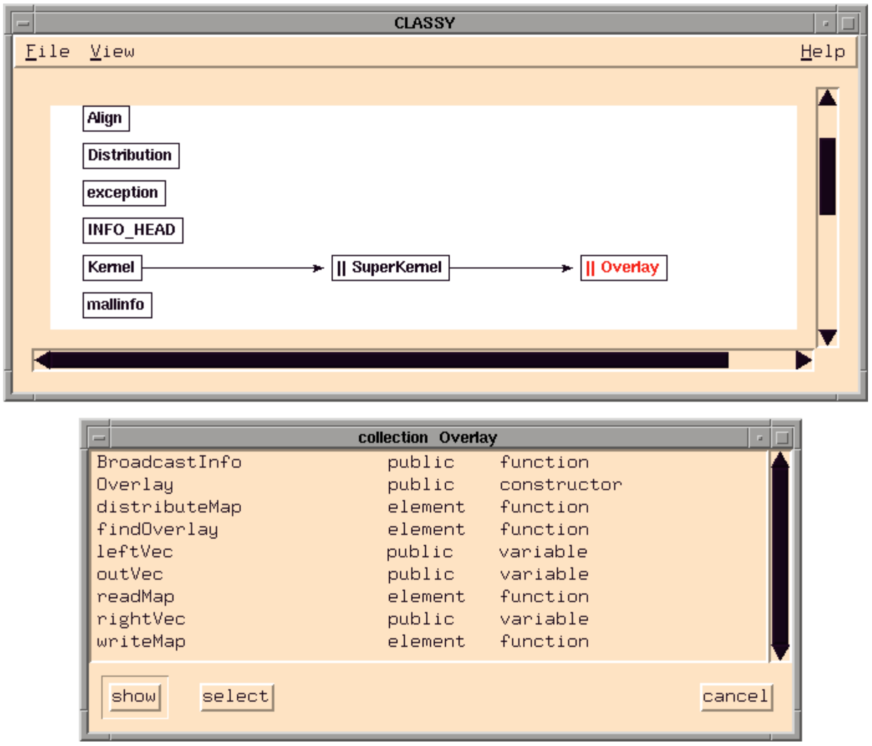


Figure 1.11: CLASSY

function. As can be seen, the pC++ polygon overlay algorithm defines its collection class by subclassing from the predefined collection `Superkernel`.

1.5.3 Dynamic Analysis Tools

$\mathcal{T}$'s dynamic analysis tools help users relate dynamic measurement results to their original pC++ programs by presenting results in terms of pC++ language objects. To show how, this section shows the results of some experiments done on an 8-processor SGI PowerChallenge with 512 MByte of memory. As stated in Section ??, the pC++ implementation of the polygon overlay problem is "embarrassingly parallel", so a dynamic analysis of it is not particularly exciting. We therefore show the dynamic behavior of the whole application, including its input and output phases.

The main window of the RACY profiling tool (Figure 1.12) gives a quick overview of the application's performance by summarizing both function

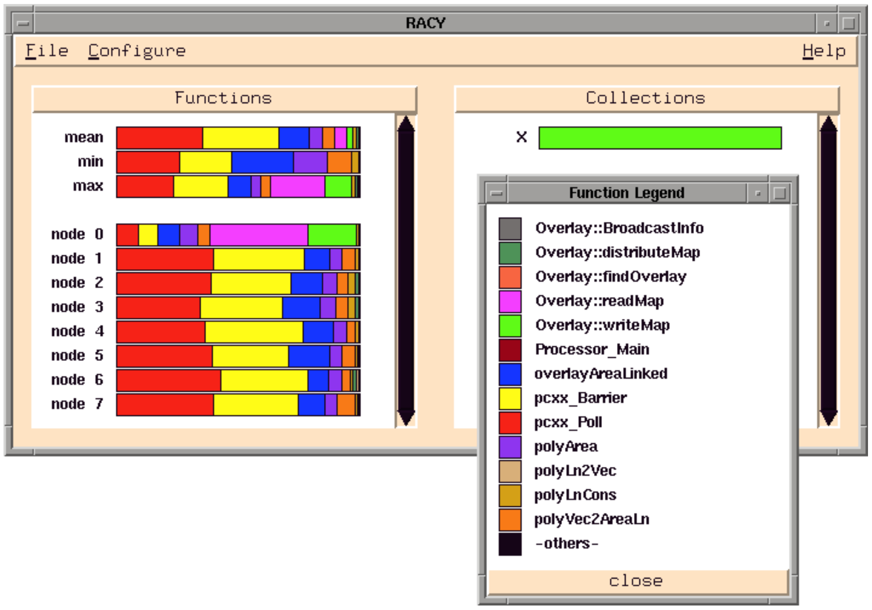


Figure 1.12: RACY Main Window

and collection access profile data. These summary displays allow us to make three important observations about the pC++ polygon overlay program:

1. Processor 0 has a different behavior than all the other nodes. A closer look using the function legend reveals that this is because node 0 is doing all of the programs' I/O (in `Overlay::readMap` and `Overlay::writeMap`).

2. The other processors spent about two thirds of their time waiting (in pC++ runtime system functions `pcxx_Barrier` and `pcxx_Poll`).

3. The good speedup of the main algorithm (`Overlay::findOverlay`) has a simple explanation: from the collection summary we can see that processes make only local accesses to the distributed collection `X`, which holds the map data. This means that there is no communication or synchronization in this part of the program.

To investigate the program further, we bring up node profiles for node 0 and node 1 (as a representative for the other nodes) by clicking on the labels in the function summary display (Figure 1.13). As the behavior of

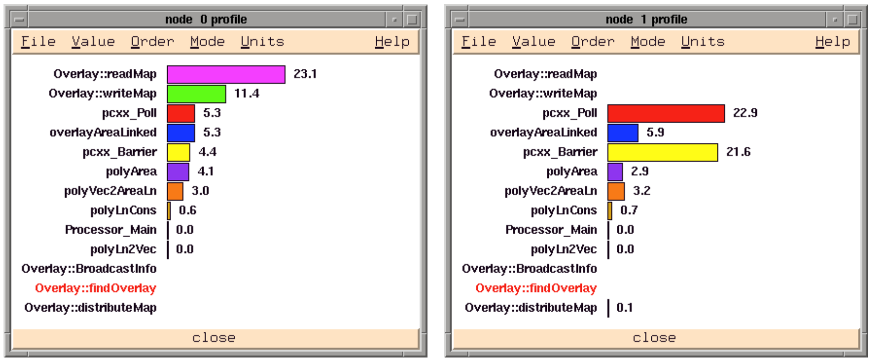


Figure 1.13: RACY Node Profile (showing seconds)

node 0 is different from that of the others, comparing the functions implementing polygon overlay is misleading if we display the execution times as percentages. We therefore configure the displays using the **Mode** and **Units** menus to show the time spent in the functions in seconds. Comparing the functions that actually implement overlay (`overlayAreaLinked`, `polyArea`, `polyVec2AreaLn`, `polyLnCons`, and `polyLn2Vec`), we can now see that they use approximately the same time.

We can make this observation even easier if we use the **Value** menu to configure the displays[3] to show the execution time including all children (Figure 1.14). We now only have to compare the execution time for `Overlay::findOverlay`.

We can use also a function profile display to compare the performance of a specific function on all nodes simply by clicking on a function name or the bar representing the function. Figure 1.16 shows the time spent in `Overlay::findOverlay` and its children on the different nodes used for the execution of the program.

This information lets us hypothesize that most of the waiting on processors 1 to 7 happens during the I/O phase on processor 0. Simple profiling does not allow us to verify our hypothesis, but we can do this very easily using event tracing.

After recompiling the program for tracing and executing it, we have an event trace for each node. Once these are merged, we can use EASY to look at the dynamic execution behavior of our application (Figure 1.18). We easily see that processors 1 to 7 are waiting in a barrier while node 0 is performing I/O (in `Overlay::readMap` and `Overlay::writeMap`). This

[3] Note that we do not have to do this for every node profile window. In using the `Configure` menu in the RACY main window, we can change all displays at once.

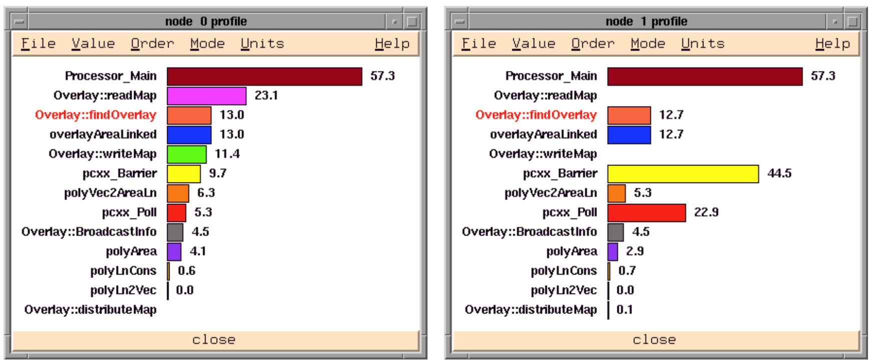


Figure 1.14: RACY Node Profile (showing total seconds)

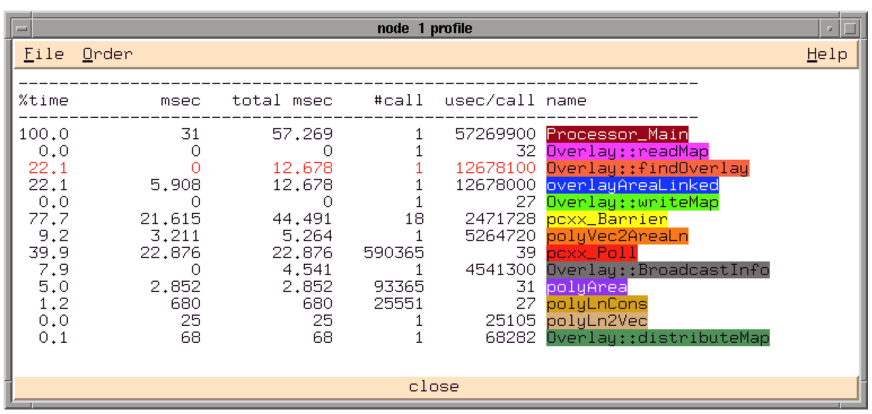


Figure 1.15: RACY Text Node Profile

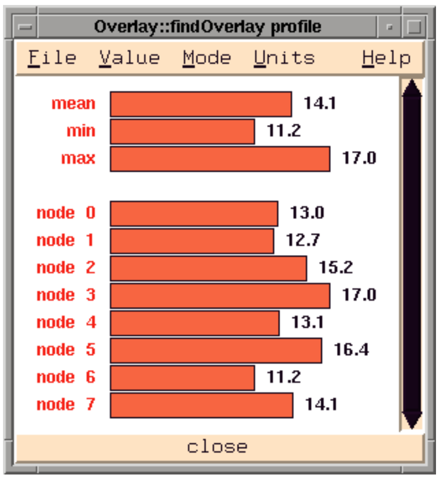


Figure 1.16: RACY Function Profile

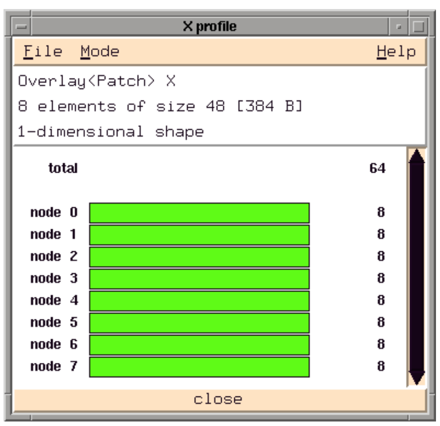


Figure 1.17: RACY Collection Profile

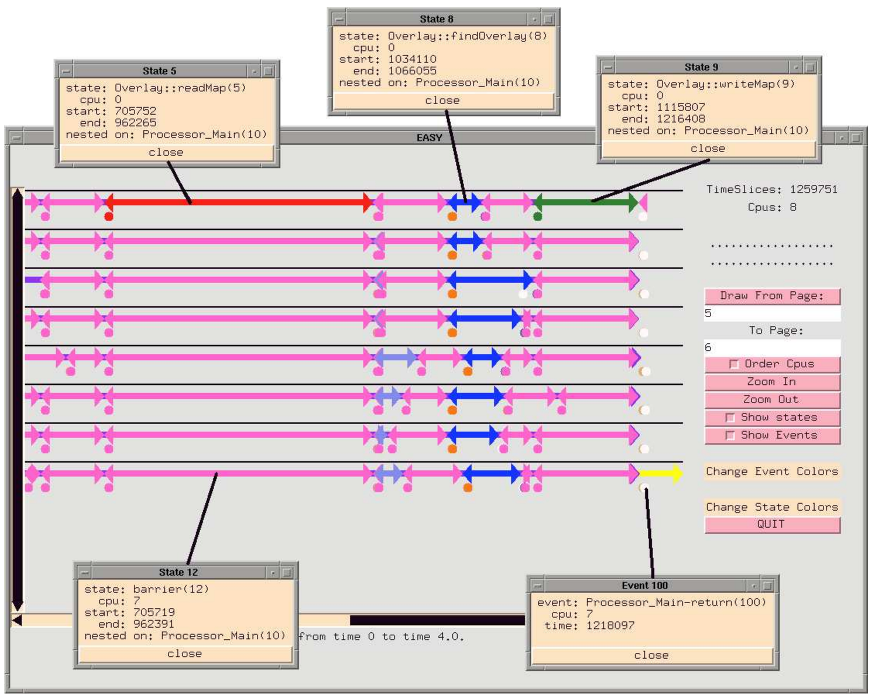


Figure 1.18: EASY

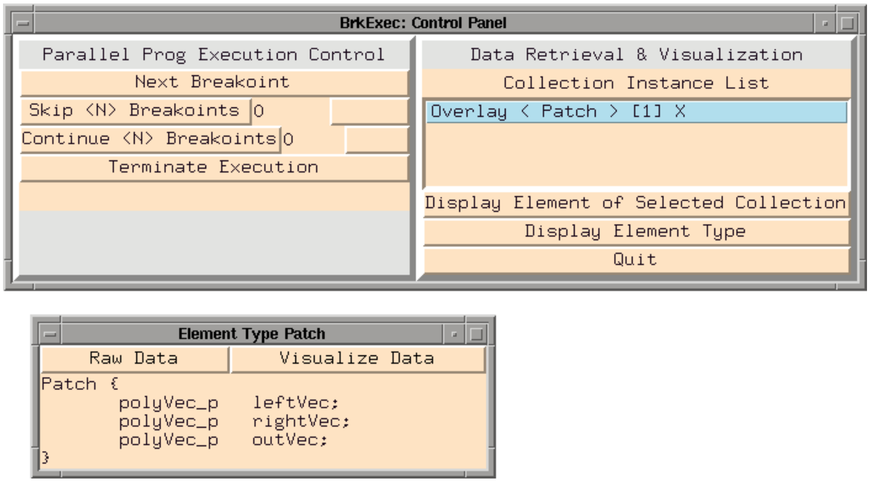


Figure 1.19: BREEZY

confirms the hypothesis we made after profiling. We can also see the typical SPMD behavior of pC++ programs. Execution of a compiler-inserted barrier after each call to a `MethodOfElement` method can be spotted by the vertical alignment of the right ends of the arrows representing the pC++ runtime system function `pcxx_Barrier`.

It is interesting to note that we measured the same performance behavior for input sizes ranging from 100 to 100,000 polygons. The ratio between the time needed for I/O and the time used for finding the overlay was roughly constant, resulting in the same general execution behavior.

The $\mathcal{T}$ barrier breakpoint debugger, BREEZY, allows users to control a pC++ program running on a parallel machine from a remote workstation. Unfortunately, the polygon overlay example is too simple to allow a full demonstration of BREEZY's functionality.

Having used COSY to compile the program for breakpointing, executing it automatically launches the BREEZY main control panel (Figure 1.19). The left side allows the user to select the next breakpoint (i.e., barrier) or to terminate execution. Every time the program is stopped at a barrier, the display of active collections is updated. In our example, there is only one active collection instance, `Overlay<Patch> X`. Selecting the collection results in a window showing the type of an element. Users can select one or more fields of the element class, and retrieve data from these fields to pipe into a visualization program.

Because the pC++ polygon overlay program re-uses the original ANSI C code to perform local node calculations, the (`Overlay<Patch> X`) collection elements only contain pointers to locally-allocated map data. BREEZY is currently unable to reference such data; this feature will be added in the next $\mathcal{T}$ release.

1.5.4 Performance Extrapolation

The use of EXTRAP/SPEEDY to estimate the performance of the polygon overlap problem proved to be more interesting than trace analysis. We used a Hewlett-Packard workstation to predict the performance of the polygon overlay code on our SGI PowerChallenge machine. We ran an n-threaded version of the code ($n = 1, 2, 4, 8, 16$) on our workstation, collecting event traces as described in Section 1.4.3. We then set execution environment parameters to correspond to the SGI machine and observed the performance behavior. The maps used in these experiments had approximately 25,000 polygons.

The SPEEDY main control window (Figure 1.20) is launched by clicking the SPEEDY button on the $\mathcal{T}$ main control panel. We build a trace-generating pC++ executable by selecting the `Compile` button of the main SPEEDY window; SPEEDY then uses COSY to execute the necessary commands. We then specify the parameters for the experimental run. Clicking on `varying parameter 1` gives us a parameter menu, from which select and modify `Number of Processors`.

The SPEEDY parameter viewer (Figure 1.21) is used to specify the necessary parameters for the EXTRAP simulation phase of the experiment. As the `Overlay:findOverlay` code in the pC++ implementation does not involve any communication or synchronization, the only significant one of the 25 EXTRAP parameters is `MipsRatio`, which is the relative speed of the CPU of the target machine (SGI) to that of the simulation machine (HP). We ran the sequential polygon overlay code on both platforms to determine this ratio. Because the polygon overlay code involves only integer arithmetic, the time on the 99MHz HP-PA chip (1.68 seconds) was close to that on the 75MHz R8000 (1.53 seconds). We therefore set `MipsRatio` to `0.91`.

Clicking SPEEDY's `Run Experiment` button starts the experiment. After each iteration of the extrapolation, the execution time graph in the SPEEDY main window is updated, which allows the user to see, and control, the evolution of the experiment. This is important for long-running experiments: if something goes wrong, it can be stopped early. Clicking on the data points in the graph displays individual values. A table can also be generated (Figure 1.22b). If the varying parameter is `Number of`

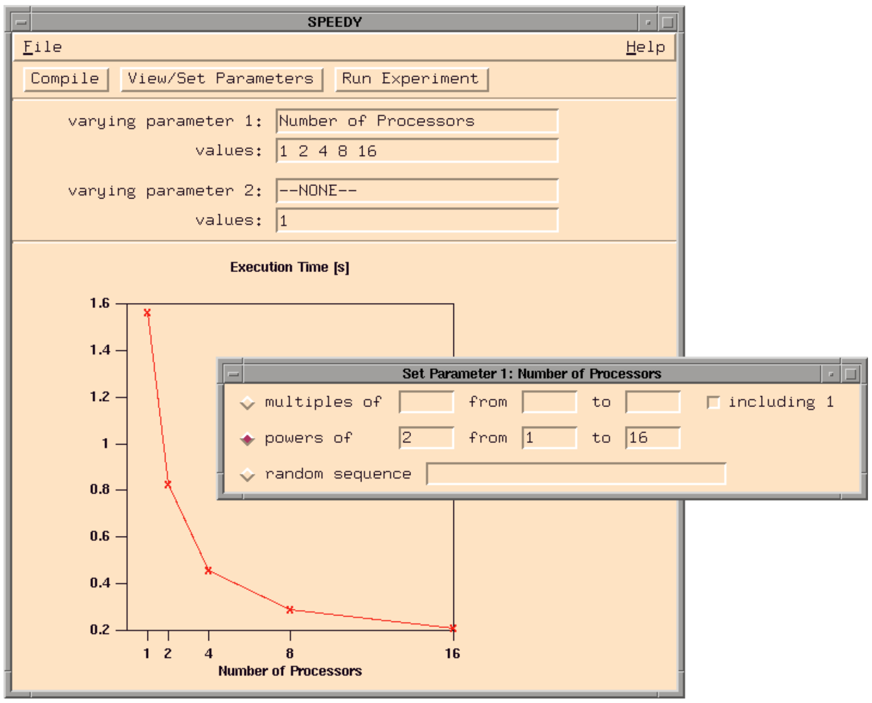


Figure 1.20: SPEEDY Main Window

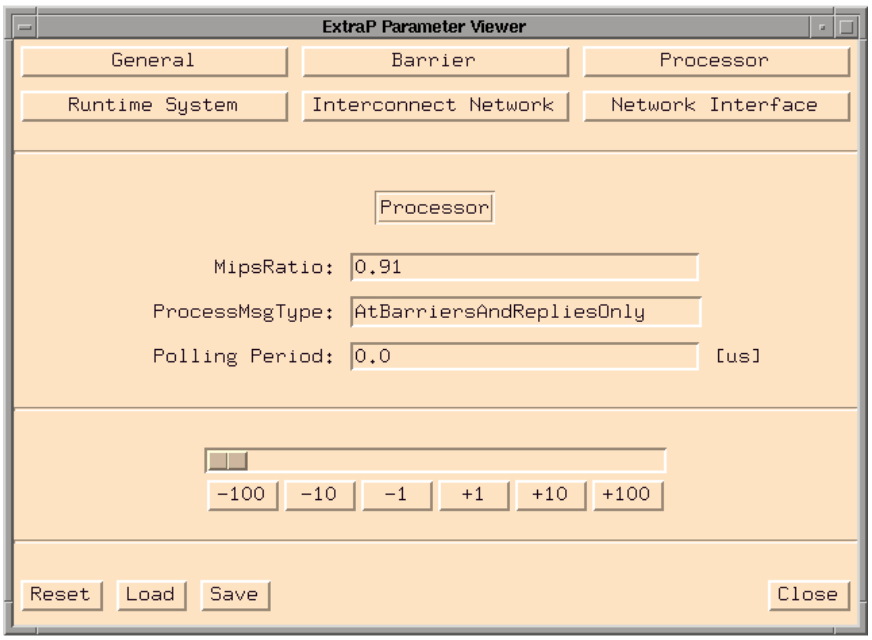
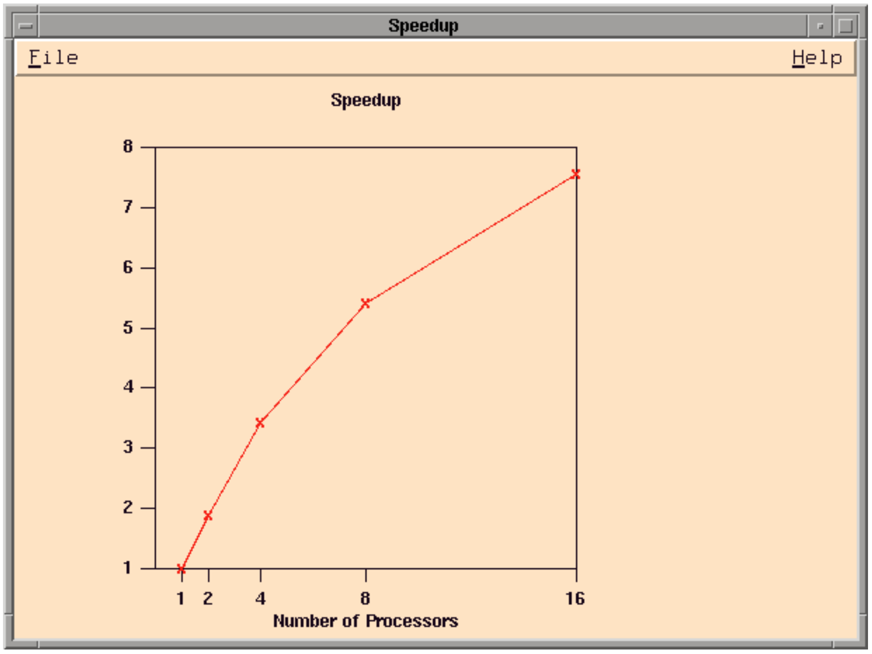
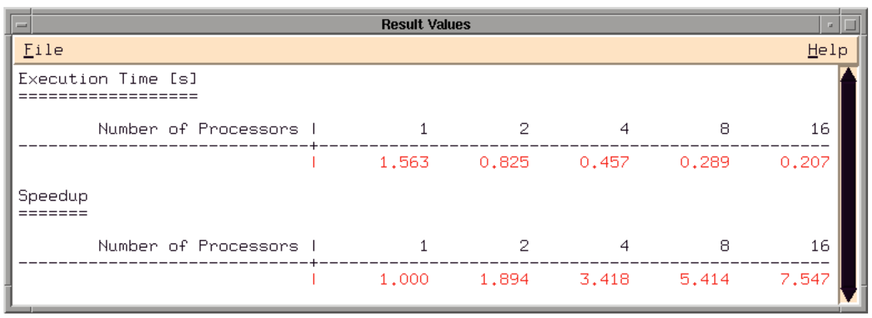


Figure 1.21: SPEEDY Parameter Viewer



(a) Speedup Display



Execution Time [s]

Number of Processors	1	2	4	8	16
	1.563	0.825	0.457	0.289	0.207

Speedup

Number of Processors	1	2	4	8	16
	1.000	1.894	3.418	5.414	7.547

(b) Result Values Viewer

Figure 1.22: SPEEDY Result Values Windows

`Processors`, as in our example, a speedup display is also available (Figure 1.22a).

SPEEDY uses one of the pC++ runtime system timers to determine which part(s) of the program to measure. This allows the user to choose whether to measure the whole program or only parts of it during a performance extrapolation experiment. In our example, we put timer calls around the `Overlay::findOverlay` method only. The results show almost linear speedup for this part of the code.

	Number of Processors				
	1	2	4	8	16
estimated	1.56	0.83	0.46	0.29	0.21
measured	1.38	0.84	0.51	0.39	–

Table 1.2: Execution Time in Seconds

As Speedy is fully integrated with $\mathcal{T}$, users can employ other $\mathcal{T}$ tools to verify extrapolation results, or compare them with actual measurements. Table 1.2 lists both measured and estimated execution times for our SGI PowerChallenge. As can be seen, the two sets of values are reasonably close. We also used the extrapolation to estimate the performance of the code on a 16 processor machine, twice as large as actually available.

1.6 Critique

In this section, we evaluate $\mathcal{T}$ based on its implementation on a large number of parallel platforms.

A parallel programming environment should support the full development circle.

The $\mathcal{T}$ environment currently supports compilation and execution control, static and dynamic program and performance analysis, debugging, and performance extrapolation. One area which is not yet supported is program development. Currently, we are working on an editor which would be integrated with the other $\mathcal{T}$ tools. One would use the static browsers to "jump" to a function, method, or class in the editor. If an error occurred during compilation with Cosy, clicking on the error message would jump to the corresponding source code line.

Also, $\mathcal{T}$ currently only supports function and class symbol lookup. We are planning to enhance the $\mathcal{T}$ browsers to support generic symbol browsing. This will require support for portable access to symbol table information, and tracking symbols between the parallel language level and the intermediate (compiler-generated) language representations.

Debugging for parallel programs is very important. The functionality of the BREEZY debugger is too restricted. We are currently implementing a new event- and state-based debugging interface. It will form the basis for a number of high-level debugging tools, including traditional debuggers, data extraction and visualization tools, and interactive profiling tools, as well as for novel application-specific debuggers. It will be realized as a very-high-level multithreaded language on top of a simple but general event-based debugging API.

Portability is a design issue. Portability is difficult to achieve, especially on parallel machines, where operating systems and C++ compilers are not as standard as they appear. Even simple things like getting a program to start executing in parallel is different on all machines. Portability must be considered from the very beginnings of a design; it cannot be achieved by first implementing a prototype for a specific platform, and then trying to port that prototype to different platforms. We believe we have solved these problem in $\mathcal{T}$, which, like pC++, runs on every major commercial parallel computer and UNIX workstation and works with every major C++ compiler. This was achieved through a combination of software engineering methods and meticulous attention to detail in our initial designs.

C++ is a very complex language. Writing commercial-quality tools for a C++-based parallel language as a University research project is almost impossible. For example, even simple things like a generation of the application's callgraph is complex, in comparison to older languages like Fortran or C because of the need to handle constructors and destructor hierarchies, operators, virtual functions, etc.

1.7 Conclusion and Future Work

The current $\mathcal{T}$ system has been highly successful in meeting the program analysis requirements of the pC++ language system. However, $\mathcal{T}$ also established a methodology and architecture for building program analysis environments that we hope to extend in three ways:

Support a wider range of programming models. $\mathcal{T}$'s tools currently focus on the data-parallel style of programming embodied in pC++ and HPF. That style makes program analysis relatively easy: data-parallel programs typically have simple, uniform communication patterns and frequent, global synchronization points. Once the model is relaxed to allow task parallelism or concurrent composition of data-parallel operations, program analysis becomes harder. Tools must contend with asynchrony, irreproducible behavior, the lack of consistent, global states, and complex patterns of process interaction.

To support these programming models, $\mathcal{T}$ will be extended to incorporate a replay mechanism. This will make it possible to repeat executions, instrumenting the code to any required level of detail. It will also make it possible to decide *post mortem* whether more sophisticated models of observability are needed. All existing $\mathcal{T}$ tools will be modified to work transparently during replay.

Replay, however, is only one approach, and is quite expensive in tracing overhead. $\mathcal{T}$ should also support more limited models of observability, giving users a choice of a range of tools. For example, one level of support might provide tracing facilities sufficient for replay, another sufficient for animation of specified data structures, another for logging procedure calls, etc. As a consequence, BREEZY (or its successor) will have to be updated to support multiple notions of a breakpoint. The user's choice of tools and options will determine the requirements for observability that are automatically supported with instrumentation.

Increase functionality. The generalization of programming model will require new, more powerful tools, particularly for debugging. Programs with complex inter-process interactions will require a multi-level debugging strategy, in which event-based techniques are initially used to find gross patterns of process interactions, and state-based techniques predominate after the focus of attention has been narrowed to a single process or small set of processes [6]. Initial use of event-based techniques focuses the user's attention on manageable portions of the state space and provides the basis for establishing consistent, meaningful, global breakpoints. Event-based techniques can incorporate replay mechanisms that support reproducible execution and logical time transformations to filter out perturbations due to asynchrony.

State-based techniques, on the other hand, allow the user to examine an execution to an arbitrary level of detail, and often make it easier to relate errors to source code. The event- and state-based tools that we are developing will be interoperable. Our current prototype demonstrates that this interaction can be quite powerful: it allows the user to set consistent breakpoints that are meaningful in the context of her ongoing event-based analysis. Often, these breakpoints would have been difficult or even impossible to set with conventional mechanisms [6].

We will need to develop appropriate abstractions to include event-based tools in our environment. Most existing tools operate at a very low level, basing their models on explicit read and write operations that are not meaningful to the pC++ or HPF programmer.

Maintain tight integration with the language system despite increasingly aggressive program transformations and optimizations. Finding appropriate abstractions for event-based tools is just one instance of a more general problem for parallel program analysis environments: the trend toward higher levels of programming abstraction coupled with generation of ever-more-efficient target code, means that tools must become increasingly sophisticated if they are to relate execution to source code. The current implementation of pC++ does not optimize aggressively, but as the next generation parallel C++ language system, HPC++, is developed, $\mathcal{T}$ will have to provide more assistance in maintaining the source/execution correspondence. Our approach will be similar to that of [1], in which the compiler provides performance tools with extensive information on the mapping between source and SPMD codes. For debugging, however, we will have to go further, enabling the tools to interpret not just performance statistics, but detailed data manipulations and control flow in terms of the initial, high-level program.

Bibliography

[1] Vikram S. Adve, John Mellor-Crummey, Mark Anderson, Ken Kennedy, Jhy-Chun Wang, and Daniel A. Reed. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing'95*, December 1995.

[2] Alfred Arnold, Ulrich Deter, and Wolfgang E. Nagel. Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding. In Bob Winget and Karen Winget, editors, *Proceedings Cray User Group Meeting, Spring 1995*, pages 252–258, Denver, Colorado, March 1995.

[3] Francois Bodin, Pete Beckman, Dennis Gannon, Jake Gotwals, S. Narayana, Suvas Srinivas, and Beata Winnicka. Sage++: An Object Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In ???, editor, *Proceedings OONSKI '94*. ???, 1994.

[4] Darryl Brown, Allen Malony, and Bernd Mohr. Language-based Parallel Program Interaction: the Breezy Approach. In *Proceedings of the International Conference on High Performance Computing (HiPC'95)*, New Delhi, India, December 1995. IEEE Computer Society, Tata McGraw-Hill.

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

[6] Joydip Kundu and Janice E. Cuny. The Integration of Event- and State-Based Debugging in Ariadne. In Constantine Polychronopoulos, editor, *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, pages 130–134. CRC Press, August 1995.

[7] Allen Malony, Bernd Mohr, Pete Beckman, Dennis Gannon, Shelby Yang, and Francois Bodin. Performance Analysis of pC++: A Portable

Data-Parallel Programming System for Scalable Parallel Computers. In Howard Jay Siegel, editor, *Proceedings of the 8th International Parallel Processing Symbosium (IPPS)*, pages 75–85, Cancun, Mexico, April 1994. IEEE Computer Society Press.

[8] Bernd Mohr. Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible? In J. J. Dongarra and B. Tourancheau, editors, *Proceedings of the CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing*, volume 6 of *Advances in Parallel Computing*, pages 103–24. Elsevier, September 1992.

[9] Bernd Mohr, Darryl Brown, and Allen Malony. TAU: A Portable Parallel Program Analysis Environment for pC++. In Bruno Buchberger and Jens Volkert, editors, *Proceedings of CONPAR 94–VAPP VI*, volume 854 of *Lecture Notes in Computer Science*, pages 29–40. Springer-Verlag, September 1994.

[10] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Version 1.2 edition, December 1993.

[11] John Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[12] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyasta, T. Birkett, D. W. Jensen, A. A. Nazief, and B. K. Totty. Scalable Performance Environments for Parallel Systems. In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 562–69. IEEE Computer Society Press, 1991.

[13] Kesavan Shanmugam. Performance Extrapolation of Parallel Programs. Master's thesis, University of Oregon, Department of Computer and Information Science, June 1994.

[14] Kesavan Shanmugam, Allen Malony, and Bernd Mohr. Performance Extrapolation of Parallel Programs. Technical Report CIS-TR-95-14, University of Oregon, Department of Computer and Information Science, May 1995.

[15] Kesavan Shanmugam, Allen Malony, and Bernd Mohr. Speedy: An Integrated Performance Extrapolation Tool for pC++ Programs. In *Proceedings of the Joint Conference PERFORMANCE TOOLS'95 and MMB'95*, September 1995.

[16] Shanmugam, Kesavan and Malony, Allen. Performance Extrapolation of Parallel Programs. In Constantine Polychronopoulos, editor, *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, volume II Software, pages 117–20. CRC Press, August 1995.