AFRL-IF-RS-TR-1998-105
Final Technical Report
June 1998

LANGUAGES, LIBRARIES AND PERFORMANCE EVALUATION TOOLS FOR SCALABLE PARALLEL SYSTEMS

Indiana University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. AO67

19980727 186

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

APPROVED: [illegible]

RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: [illegible]

NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

LANGUAGES, LIBRARIES AND PERFORMANCE EVALUATION TOOLS FOR SCALABLE PARALLEL SYSTEMS

Dennis Gannon
Allen Malony
Michael Wolfe

Contractor:	Indiana University
Contract Number:	F30602-92-C-0135
Effective Date of Contract:	13 August 1992
Contract Expiration Date:	13 August 1997
Program Code Number:	A0670001
Short Title of Work:	Languages, Libraries and Performance Evaluation Tools for Scalable Parallel Systems
Period of Work Covered:	Aug 92 - Aug 97
Principal Investigator:	Dennis Gannon
Phone:	(812) 855-5184
AFRL Project Engineer:	Raymond A. Liuzzi
Phone:	(315) 330-3577

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Raymond A. Liuzzi, AFRL/IFTB, 525 Brooks Road, Rome, NY, 13441-4505.

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 1998	**3. REPORT TYPE AND DATES COVERED** Final Aug 92 - Aug 97

4. TITLE AND SUBTITLE

LANGUAGES, LIBRARIES AND PERFORMANCE EVALUATION TOOLS FOR SCALABLE PARALLEL SYSTEMS

5. FUNDING NUMBERS

C	-	F30602-92-C-0135
PE	-	62301E
PR	-	A067
TA	-	00
WU	-	01

6. AUTHOR(S)

Dennis Gannon, Allen Malony, and Michael Wolfe

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

Indiana University
Computer Science Department
Bloomington, Monroe County, IN 47405-4101

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

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

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Air Force Research Laboratory/IFTB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1998-105

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Raymond A. Liuzzi/IFTB/(315) 330-3577

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The primary objective of this project has been to develop a new programming language, compiler and programming environment, called pC++, that is based on a simple extension to C++ to support the development of software for Massively Parallel Processing (MPP) computer systems. This tool supports a platform for parallel object-oriented software capable of running without modification on all commercial Multiple-Instruction-Multiple-Data (MIMD) systems; an interface to Single Program Multiple Data (SPMD) libraries such as ScaLapack++, A++ and POOMA; an interface to High Performance Fortran (HPF); an interface to control-parallel C++-based languages such as CC++; a way to exploit parallel I/O systems and persistent object databases; and a complete programming environment. PC++ has been implemented on a wide range of commercially available parallel systems including the Thinking Machines CM-5, the Intel Paragon, the SGI PowerChallenge, the IBM SP-2 and the Cray T3-D. One of the most interesting by-products of the pC++ project has been a language preprocessor toolkit called Sage++. This toolkit has been extended in a variety of ways and is used for a large number of applications, including the TAU programming environment. There have been four major components to C++ technology. The first component is the pC++ language. The mot important of these are applications that are part of the NSF NCSA Alliance. The second component of technology is the Sage++ compiler toolkit that was built for pC++. The third component of technology transfer is the Tulip runtime system, the ASCI Blue Mountain system. The fourth component of the technology is the TAU programming environment tools and will be the standard foundation for HPC++ tools.

14. SUBJECT TERMS Computers, Software, Programming, Parallel Systems	**15. NUMBER OF PAGES** 102
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

1 Introduction

1.1 Executive Summary.

The primary objective of this project has been to develop a new programming language, compiler and programming environment, called pC++, that would be based on a simple extension to C++ to support the development of software for Massively Parallel Processing (MPP) computer systems. More specifically, the tool should support:

- a platform for parallel object-oriented software capable of running without modification on all commercial Multiple-Instruction-Multiple-Data(MIMD) systems;
- an interface to Single Program Multiple Data (SPMD) libraries such as ScaLapack++, A++ and POOMA;
- an interface to High Performance Fortran (HPF);
- an interface to control-parallel C++-based languages such as CC++;
- a way to exploit parallel I/O systems and persistent object databases; and
- a complete programming environment including all the tools that users of conventional C++ systems expect, as well as tools for parallel performance analysis and debugging.

We do not think that pC++, or other object-oriented parallel programming languages, should be viewed as replacements for Fortran 90 or HPF. Rather, object-oriented parallelism should be used to express those types of parallelism that cannot easily be expressed in these languages.

pC++ is based on a *concurrent aggregate* model of data parallelism. This means that a pC++ program consists of a single main thread of control from which parallel operations are applied to *collections* of objects. Each object in a collection is an instance of an *element* class. pC++ has two basic extensions to the C++ language: a mechanism to describe how operations can be invoked over a set of objects in parallel, and a mechanism to refer to individual objects and subsets of objects in a collection.

pC++ has been implemented on a wide range of commercially available parallel systems including the Thinking Machines CM-5, the Intel Paragon, the SGI PowerChallenge, the IBM SP-2 and the Cray T3-D. Our primary experience with testing the pC++ ideas on large scale problems has come from our involvement with the NSF Grand Challenge Cosmology Consortium GC^3. This report describes some of these applications.

One of the most interesting by-products of the pC++ project has been a language preprocessor toolkit called Sage++. This toolkit has been extended in a variety of ways and is used for a large number of applications, including the TAU programming environment described in the next section of this report.

Most of the key ideas and technology developed for pC++ has been integrated in the DARPA HPC++ project which is described in the technology transfer section of this report.

1.1.1 Project Accomplishments

In this section we describe the three primary technical accomplishments of this project: the pC++ language and compiler, the Sage++ compiler technology and the TAU programming environment.

The pC++ Compiler and Library Effort The final version of pC++ was made available in Q4 of 1996 and it can be access by the World Wide Web at ftp://ftp.extreme.indiana.edu/pub/sage/ in file pc++sage++2.0.tar.Z. This file contains the entire distribution of all the software described here. In addition, a Web based user's guide is available at http:// www.extreme. indiana.edu/sage/ pcxx_ug/pcxx_ug.html.

The compiler consists of a pC++ to C++ translator that works with any conventional C++ system and our parallel runtime library TULIP. In other words, pC++ programmers write pC++ parallel application code and the translator converts their program files to a set of C++ program files that make calls to our special parallel execution library. The system automatically compiles the C++ programs and links the libraries.

pC++ programs are completely portable. That is, if a program is written with pure pC++ it will compile for any of the parallel architectures supported by the last release (SGI, Cray T3D, IBM SP2, Intel Paragon, TM CM-5). Furthermore the performance of a well tuned pC++ program matches HPF and Fortran plus message-passing.

Example Performance Results pC++ has been tested on a number of examples. However, one of the most impressive examples was a simulation of the collision of two Galaxies involving an N-Body computation with 40 million bodies. This project was awarded the prize for "Best Demonstration of Heterogeneous Computing" at the I-Way demonstration at Supercomputing 95.

The core of this computation was a parallel simulation designed as part of the GC^3 NSF Grand Challenge Project. The simulation, known as a Self Consistent Field (SCF) computation was written in pC++, CM Fortran and Fortran+MPI. For the large simulation the pC++ and Fortran+MPI versions were linked together over two supercomputers with 512 processors each. To illustrate the scalability of pC++ performance we tested the pC++ SCF code on a smaller example data set on a number of different machines.

Our experiments with the pC++ SCF code were conducted on a Thinking Machines CM-5, an Intel Paragon, an SGI Power Challenge, an IBM SP-2, and

Platform	Number of Processors			
	8	16	32	64
Cray T3D		223.0	115.3	
Intel Paragon		667.3	332.5	168.5
IBM SP-2		186.9	103.5	
Power Challenge	116.9	58.6		
CM-5 (pC++)			45.8	
CM-5 (CM Fortran)			50.3	

Table 1: SCF code execution time, in seconds, for evolving a 51,200 particle stellar system for 100 time steps.

a Cray T3D. For comparison, we also ran the CM Fortran SCF code on the CM-5. 51,200 particles were used for the simulation. The system was allowed to evolve for 100 time steps. The results of these experiments are listed in the table below for the five different platforms and three configurations of numbers of processors. (In some cases, only smaller numbers of processors were available. In other cases, only 64 processors were available.)

The Sage++ Compiler Tools The Sage++ compiler tool kit is one of the most important technology spin-off products of this contract. It is now widely used in university research as well as commercial products (see technology transfer.)

Sage++ consists of a set of tools to translate Fortran, C++, pC++ and Corba IDL programs into a C++ object tree form. It is then possible for a programmer to write a tool that can read this object tree representation, analyze it, transform it, and then translate the modified object tree back into the original source language. The object tree representation of a program contains complete information about program variables and types as well as all statements and expressions. Consequently, tools that are built with Sage++ have complete access to the semantics of the program and can modify them in any way.

The types of applications that have been built with Sage++ include the following major efforts.

- the pC++ compiler.
- the Argonne Fortran-M parallel programming system.
- the INRIA (France) Fortran-S parallel Fortran system.
- the TAU programming environment (see below).
- the Argonne ADIC C program Automatic program differentiation system.

- the Syracuse HPF compiler.
- the NASA AIMS parallel performance tools system.
- the PARDIS parallel CORBA system.

All of these systems are currently in public distribution. In addition, IBM has licensed the Sage++ software for use in internal research projects and two small companies have acquired licenses to distribute sage as part of commercial products.

The complete source code directory for Sage++ can be found in the pC++ distribution pc++sage++2.0.tar.Z described in the previous section. In addition, a complete, interactive users guide is available on line at http://www.extreme.indiana.edu/sage/sagexx_html/sagexx_ug_toc.html.

However, two years of use has demonstrated to us the need for a number of important improvements in Sage. In collaboration with Caltech, we have undertaken a complete redesign of Sage++. The new system, Sage2 will be based on a commercial C++ front-end provided by Edison Design Group, Inc. (EDG). We expect that the resulting system will be a standard for the design of programming tools for C++, Fortran, Java and CORBA IDL. This is discussed in greater detail in the Technology Transfer section of this report.

The Tulip Runtime System Tulip a parallel run-time system (RTS) that is used as part of the pC++ parallel programming language. The RTS has been implemented on a variety of scalable, MPP computers including the IBM SP2, Intel Paragon, Meiko CS2, SGI Power Challenge, and Cray T3D. This system differs from other data-parallel RTS implementations; it is designed to support the operations from object-parallel programming that require remote member function execution and load and store operations on remote data. The implementation is designed to provide the thinnest possible layer atop the vendor-supplied machine interface. That thin veneer can then be used by other run-time layers to build machine independent class libraries, compiler back ends, and more sophisticated run-time support. Tulip is now the standard runtime system being used in pC++ and it is also the basis for the HPC++ runtime layer.

Tulip is now a major component of the DOE ASCI effort (See Technology Transfer section below.)

1.1.2 The Tau Program Analysis Environment

TAU (Tuning and Analysis Utilities) is a visual programming and performance analysis environment for pC++. Elements of the TAU graphical interface represent objects of the pC++ programming paradigm: collections, classes, methods, and functions. These language-level objects appear in all TAU utilities. TAU uses the Sage++ toolkit as an interface to the pC++ compiler for instrumentation and accessing properties of program objects. TAU is also integrated with the pC++ runtime system for profiling and tracing support. TAU is implemented in C and C++ and is using Tcl/Tk for graphics.

The TAU tools are implemented as graphical hypertools. While they are distinct tools, they act in concert as if they were a single application. Each tool implements some well-defined tasks. If one tool needs a feature of another one, it sends a message to the other tool requesting it (e.g., display the source code for a specific function). This design allows easy extensions.

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

The components of TAU include

- TAU (TAU Main Control Window) It allows you to start the other tools, provides on-line, hyper-text help and some global functionality like loading another .dep file (the internal program tree form for pC++ programs).

- COSY (COmpile manager Status displaY) This tool provides a user-friendly and convenient way of compiling and executing pC++ programs. Through a graphical interface, the user can first select the parallel machine on which the given application is to be compiled and run. Parameters and options for the compilation process (e.g., compile for tracing) and for the program run (e.g., activated event classes for trace recording) can be 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.

- FANCY (File ANd Class displaY) lets you browse through the files and classes used in the source text of the application, and lets you display the source text of functions, methods, or classes.

- CAGEY (CAll Graph Extended displaY) lets you browse through the static callgraph of the application.

- CLASSY (CLASS hierarchY browser) displays the class hierarchy defined in the current user application.

- RACY (Routine and data ACcess profile displaY) is a parallel profile data viewer. After compiling an application for profiling and running it, racy lets you browse through the function and collection profile data generated.

- SPEEDY (Speedup and Parallel Execution Extrapolation DisplaY) is a graphical interface to the pC++ simulation environment XtraP. It allows performance analysis and extrapolation of pC++ programs based on execution traces.
- BREEZY (BReakpoint Executive Environment for visualiZation and data displaY) is for breakpoint-based program analysis. Breezy allows a user to control the execution of a pC++ program and to view the parallel program collection data during the execution, the essential functions of a parallel debugger. The execution is controlled by manipulating program breakpoints. At these breakpoints, the program data can be displayed in a text window or visualized by a compatible visualization tool.
- CRAFTY (ContRol flow And FuncTion displaY) is a control flow graph browser. The nodes of the graphs (conditionals and basic blocks) contain the list of functions and methods which are called from this block. This tool will also allow the graphical specification of instrumentation points for profiling and tracing.

Tau source code is included with the current distribution of pC++. Complete tutorial and user guide information is available from the pC++ distribution and on line at http://www.cs.uoregon.edu/research/paracomp/proj/tau/.

1.2 Technology Transfer

There have been four major components to pC++ technology and all three are being moved into wider distribution and evolution. The first component is the pC++ language. The transition from pC++ to a working HPC++ standard has been a long goal of this effort and it is now a reality. Because the C++ language has evolved so much in the past year, it is now possible to do more in C++ without many of the language extensions that exist in pC++. This has simplified the design of HPC++ considerably. In fact, the primary data structure of pC++, the collection class type, is now seen as a Parallel Standard Library *container* class. We are in the process of converting many of the existing pC++ applications to the HPC++ syntax. The most important of these are applications that are part of the NSF NCSA Alliance which is funding the effort to make pC++/HPC++ more widely used there.

The second component of technology is the Sage++ compiler toolkit that was built for pC++. The Sage++ system has now been licensed by IBM corporation for internal research purposes. It has also been licensed by NASA Ames research laboratories for distribution as part of their Ames performance analysis tools. In addition, Amerinex corporation has licensed Sage++ for commercial distribution as a component in their product line. Sage2 development is now very active as part of the HPC++ project and it should be released by the end of 1997. We expect that Sage2 will be even more widely used because it incorporates an existing, commercial quality language parser for C++.

The third component of our technology transfer is the Tulip runtime system. Tulip has now been adopted by the DOE ASCI program as one of the core runtime kernels for the ASCI Blue Mountain system. Consequently, Tulip will be an important component in software design for the stockpile maintenance project. Because of this key role Tulip will play, Los Alamos has taken over the maintenance of Tulip and is working with us on the integration of Tulip into HPC++.

The fourth component of the technology is the TAU programming environment tools. TAU has now been integrated with the Portland Group Inc, HPF compiler and will be the standard foundation for HPC++ tools. In fact the first HPC++ versions of TAU are currently in Beta test mode.

1.3 The Remainer of This Report

The remainder of this report consists of two chapters which provide a detailed discussion and analysis of the two primary deliverable technologies of this project: pC++ and the associated performance analysis toolkit TAU. These two chapters have been designed to be read as stand-alone documents. However, they are deeply connected. As an illustration of the use of PC++ we have included in the first chapter a detailed solution of a problem known as the polygon overlay problem. In the TAU chapter, we use this same computation as part of our analysis of the effective use of the TAU tools. The TAU chapter concludes with a assessment of the system at the time of the project completion. It should be noted that TAU continues to evolve and later versions now address many of the issues that are discussed in the assessment section.

A complete bibliography of related work appears at the end of this report.

2 pC++

pC++ is a data-parallel extension to C++ that is based on the concept of *collections* and *concurrent aggregates*. It is similar in many ways to newer languages like ICC++, Amelia and C** in that it is based on the application of functions to sets of objects. However, it also allows functions to be invoked on each processor to support SPMD-style libraries and it is designed to link with HPF programs. pC++ currently runs on almost all commercial massively-parallel computers, and is being used by the NSF Computational Grand Challenge Cosmology Consortium to support simulations of the evolution of the universe. In this chapter we describe the language and its performance on a variety of problems.

2.1 Introduction

The goal of the pC++ project was to design a simple extension to C++ for parallel programming that provides:

- a platform for parallel object-oriented software capable of running without modification on all commercial MIMD systems;
- an interface to Single Program Multiple Data (SPMD) libraries such as ScaLapack++ [8], A++ [21] and POOMA ;
- an interface to High Performance Fortran (HPF) [19];
- an interface to control-parallel C++-based languages such as CC++ [7] ;
- a way to exploit parallel I/O systems and persistent object databases; and
- a complete programming environment including all the tools that users of conventional C++ systems expect, as well as tools for parallel performance analysis and debugging.

We do not think that pC++, or other object-oriented parallel programming languages, should be viewed as replacements for Fortran-90 or HPF. Rather, object-oriented parallelism should be used to express those types of parallelism that cannot easily be expressed in these languages. To accomplish this, pC++ exploits the two definining characteristics of object-oriented design: encapsulation and inheritance.

pC++ is based on a *concurrent aggregate* model of data parallelism. This means that a pC++ program consists of a single main thread of control from which parallel operations are applied to *collections* of objects. Each object in a collection is an instance of an *element* class. pC++ has two basic extensions to the C++ language: a mechanism to describe how operations can be invoked over a set of objects in parallel, and a mechanism to refer to individual objects and subsets of objects in a collection.

pC++ has been implemented on a wide range of commercially available parallel systems; we describe its performance on such platforms later in this chapter. Other examples of pC++ programs and performance can be found in [12, 13, 10, 4, 14]. Our primary experience with testing the pC++ ideas on large scale problems has come from our involvement with the NSF Grand Challenge Cosmology Consortium GC^3. This chapter describes some of these applications. We also discuss two libraries that support parallel I/O and persistent objects in pC++ programs.

One of the most interesting by-products of the pC++ project has been a language preprocessor toolkit called Sage++ [3]. This toolkit has been extended in a variety of ways and is used for a large number of applications, including the TAU programming environment described in the TAU Chapter.

2.2 History

In 1984, the parallel programming research group at Indiana University, working with the Center for Supercomputing Research and Development (CSRD) at the University of Illinois, developed an extension to the C programming language called Vector Parallel C (VPC) [11]. VPC used parallel loops for spawning new threads of control, a vector notation similar to Fortran-90 for data-parallel operations, and assumed a shared memory model of execution.

By 1986, we had become interested in distributed memory multicomputers, and decided to build a new system based on object-oriented design ideas. Our goal was to implement parallel control mechanisms by applying member functions to sets of objects. The first problem to be solved was how to describe a generic set of objects in C++. At the time, the C++ template mechanism was not yet a complete proposal to the C++ standards committee, although early public documents such as [31] guided our thinking.

Even had they existed, templates would not have solved all of our problems. To see why, consider the following definition of a set of objects of type T derived from a templatized container class Set:

```
Set<T> S;
```

Suppose that the set element type T takes the form:

```
class T {
 public:
  void foo();
};
```

Our desire was to be able to apply the member foo() to the entire set S in parallel with the expression S.foo(). Unfortunately, this could not be done using the standard overloading and inheritance mechanisms of C++. Furthermore, because there were no implementations of templates in C++ at that time, we decided to add an extension to pC++ to represent a type of class called a *collection*. Each collection had one built-in "template" parameter called ElementType. To simplify the compiler, we put the mechanisms for managing a distributed set of elements into a library called the SuperKernel collection. The way in which these collection classes are used is described in detail in the next section.

About the time that our first implementation of pC++ for shared-memory multiprocessors was complete, the HPF Forum was being established. Because HPF was also a data-parallel programming language, we were convinced that we needed to base the allocation and data distribution mechanisms for collections on distributed memory systems on the HPF model. Such a design would help make it possible to share distributed data structures with HPF implementations (although this idea has never been tested). In retrospect, we have realized that, for most users, a standard interface to single-node Fortran-90 is more important than compatibility with HPF. This is because the majority of large pC++ applications that are in production use are written with Fortran subroutines that have been scavenged from older sequential and vector versions of the application.

In 1992, ARPA provided the support for a complete redesign of pC++ and a public release. The final version of pC++ (version 2.0) will be released in early 1996. This chapter describes this new version of the language.

2.3 Overview of pC++ Version 2.0

pC++ was designed to work on both multiprocessors and multicomputers. We use the HPF model to describe the way in which an array-like data structure can be distributed over the memory hierarchy of a parallel computer. To build a collection of objects from some class type T, which is called an *element* class[1] in pC++, one needs a distribution and an alignment object. The distribution object defines a grid and a mapping from the grid to the physical processors on a parallel machine. The alignment object specifies the shape, size, and the mapping of the element objects to the grid points. In addition, a processor object of type Processors is needed to represent the set of processors available to use. For example:

```
Processors P;
Distribution D(100, &P, BLOCK);
Align A(20, "[ALIGN(X[i], D[i+10])]");
```

creates an one-dimensional grid of a size of 100 which is mapped to the processors of the machine by blocks. If there are 20 processors, grid positions 0 through 4 are mapped to processor 0, positions 5 through 9 are mapped to processor 1, etc. The alignment object aligns the logical vector X[0:19] with the grid positions D[10:29].

Given a distribution, an alignment and the class type of the element objects, it is easy to build a collection. The starting point is the SuperKernel collection

[1]In its current implementation, elements of a collection must be of the same type.

provided by the pC++ collection library. This collection is the base type for all other collections. It builds arrays of element objects and provides a global name space for the element objects. Thus, the declaration:

```
SuperKernel<T> MyCollection(&D, &A);
```

creates a collection called `MyCollection`, consisting of a set of 100 objects of type `T` distributed in the manner described above.

The most important feature of a collection is the ability to apply a function in parallel across all the element objects. For example, if `T` is defined as:

```
class T{
  ...
 public:
  void foo();
  int x, y, z;
  float bar(T &);
  ...
};
```

a parallel application of `foo()` to all elements of `MyCollection` would take the form:

```
MyCollection.foo();
```

In the case above, `foo()` has a void result, so the expression `MyCollection.foo()` has a void result as well. However, pC++ extends the type system so that, for example, if `x` is a type `int` data member of the element class, then `MyCollection.x` is an object of type `SuperKernel<Int>`, where `Int` is a library class with one integer value. The expression:

```
MyCollection.x = 2*MyCollection.y + MyCollection.z;
```

is therefore a parallel computation involving element-wise multiplication, addition and assignment on the members of each element of the collection.

Similarly, if `t` is of type `T` the expression `MyCollection.bar(t)` applies `bar(t)` to each element of the collection. The result is of type `SuperKernel<Float>`. Also, if `C` is another collection whose size is the same as `MyCollection` and whose element type is `T`, the expression

```
MyCollection.bar(C)
```

will apply `bar()` to the i^{th} element of `MyCollection` using the i^{th} element of `C` as an argument.

It is often the case that an operation must be applied to a subset of the elements of a collection. pC++ extends the Fortran-90 vector notation so that descriptors of the form `base:end:stride` can be used to select elements from a collection. For example:

```
MyCollection[0:50:2].foo()
```

will apply `foo()` to the first 25 even numbered elements of the collection.

To access an individual member of a collection, one can use the overloaded operator `()` which returns a *global* pointer to an element, i.e. a pointer that can span the entire address space of a distributed-memory machine. For example:

```
MyCollection(i)
```

returns a global pointer to the i^{th} element in the collection. In this way, any object can have a global address. The function call:

```
MyCollection(i)->foo();
```

is a remote invocation. It sends a message to the processor that contains the i^{th} element of `MyCollection`, and a thread on that processor executes the function[2]

Programmers often need to create specialized collections with properties appropriate for their particular applications. The task of building a new derived collection is almost the same as building a derived class in C++. The definition of a collection derived from `SuperKernel` takes the form:

```
Collection MyCollectionType: SuperKernel {
 public:
  // Public data members duplicated on each processor.
  // Public member functions executed in parallel on all processors.
 MethodOfElement:
  // Data members and member functions here are added to
  // the element class.
};
```

There are two types of data and member functions in a collection definition. Data and functions labeled as `MethodOfElement` represent new data members and functions that are to be added to each element class. Such member functions are invoked and executed in the same way that ordinary element class member functions are invoked and executed. Data members not labelled as `MethodOfElement` are defined once on each processor; functions not labeled `MethodOfElement` are invoked in "SPMD" mode. This is similar to the extrinsic function execution model in HPF. More precisely, pC++ has a single sequential thread of control for all operations other than collection member function calls. Collections are data aggregates that may be distributed over multiple address spaces. Invoking a collection member function that is not a `MethodOfElement` member causes a thread of control in each address space to execute the function. These new threads of execution are independent and run in parallel. The programmer is free to embed explicit communication and syncronization in these functions. The functions are barrier synchronized before control is returned to the single sequential main thread. On distributed memory systems, where there is one processor per address space, the number of threads that are running concurrently in a non-`MethodOfElement` call is one per processor. For

[2]Remote invocation of this kind is part of pC++ 2.0, and is not part supported by the current pC++ 1.0 distribution.

MethodOfElement function calls there is one function invocation per element in the collection.

2.3.1 pC++ Run-Time System

pC++ is extremely portable. It currently runs on the Cray T3D, IBM SP2, Intel Paragon, Meiko CS-2, SGI Power Challenge, TMC CM5, Convex Exemplar, and networks of workstations. The key to this portability is the simple execution model and layered run-time system. The first run-time layer is machine independent and is defined by the pC++ compiler (source-to-source translator). The compiler generates calls to the C++ class library whose interface is defined in `kernel.h`. There are two versions of this class library layer: one for SPMD execution and one for fork/join thread-based execution.

For distributed memory machines, SPMD execution is used, and the pC++ compiler converts parallel invocations such as:

```
MyColl.foo();
```

into loops over the local collection elements using the "owner computes" rule as shown below. First the data type

```
MyCollection<T> MyColl(...);
```

is converted by the compiler to an explicit C++ class

```
MyCollection_T MyColl(...);
```

and the function invocation is converted into the loop

```
for (i= MyColl->FirstLocal(); i >= 0; i = MyColl->NextLocal(i))
  MyColl(i)->foo();
pcxx_Barrier();
```

The generated loop uses the overloaded `()` operator, provided by the pC++ class library, to find the i^{th} collection element. After each processor has applied `foo()` to its local elements, a barrier synchronization between processors in initiated.

Shared memory machines can use the SPMD model shown above, or the pC++ compiler can generate a special thread-based run-time interface. More specifically, the loop above now takes the following abstract form. Let us assume that there are k processors available

```
fork_threads(k); // create or allocate k threads of execution
   // each thread executes the following
   int s = MyColl->size(), me = my_thread_id();
   for(int i = (s/k)*me; i < min(s, (s/k)*(me+1)); i++)
      MyColl(i)->foo();
join_threads(k); // wait for all thread to reach this point
                 // and terminate or suspend all but one.
```

The abstraction of work is sufficiently general to permit many different thread packages. An implementation could create a new thread for each element, i.e. let $k = s$, or use a set of k persistent threads.

2.3.2 Tulip

The next portion of the run-time system is the low-level machine-dependent layer, called Tulip. Tulip describes an abstract machine, and defines standard interfaces for basic machine services such as clocks, timers, remote service requests, and data movement. Tulip has a C interface, and has no knowledge of pC++ or the class library, which are built on top of Tulip. Therefore, wherever Tulip can be ported, pC++ can run.

Tulip has several basic abstractions:

- *Context*: An address space. A Unix process on a symmetric multiprocessor would be a single context. Lightweight threads share a context. A machine such as the SP2 can support several contexts per node.
- *LocalPointer*: A simple, untyped, memory address. A LocalPointer is valid only within the Context it was created.
- *GlobalPointer*: The tuple (Context, LocalPointer). A GlobalPointer uniquely identifies any memory address in the computational hardware.

Those abstractions are used in the following basic functions:

```
tulip_Put(tulip_GlobalPointer_t destination, char *source,
          int length, tulip_ACK_t *handle);

tulip_Get(char *destination, tulip_GlobalPointer_t source,
          int length, tulip_ACK_t *handle);

tulip_RemoteServiceRequest(int context, char *buffer,
                           int length, tulip_ACK_t *handle);
```

Put() and Get() simply move data between contexts. They are very similar to memcpy(), except destination and source are global pointers respectively. Furthermore, an acknowledge handle is provided so the status of the data transfer can be monitored. If the handle is NULL when the function is called, no acknowledgment is done. The functions are non-blocking, so that they can be easily integrated with user-level thread packages.

The remove service request mechanism provides asynchronous communication between contexts. It is particularly useful for bootstrapping, building remote procedure execution for pC++ (see section Section 2.3), and transmitting short control messages to other contexts.

The basic abstractions and functions are supported on three architectural models: shared memory, message passing, and network DMA.

The SGI Challenge and Convex Exemplar are examples of shared memory machines. The hardware maintains cache and memory consistency, and communication is done by simply sharing pointers. In this case, `Put()` and `Get()` need not be used, because those functions move data between contexts. On a shared memory machine, there is usually only one context. However, if `Put()` or `Get()` are used, they are simply a call to `memcpy()` followed by the `TRUE` acknowledge handle.

Two examples of message passing machines are the Intel Paragon and IBM SP2. Since `Put()` and `Get()` are one-sided communication primitives, and do not require synchronization, either active messages or polling loops must be used to detect when a data movement request arrives. For each `Get()`, a `recv()` is posted for the anticipated data, then a data request message is sent to the remote context (i.e. node). When the sender detects the data request message during a message poll, the data is sent to the awaiting `recv` without a buffer copy. `Put()` uses a similar mechanism, but requires an extra round trip to avoid any buffer copies. If the message is sent to the remote context "eagerly", the extra round trip latency is not incurred, but the messaging system must copy and buffer the data.

The Meiko CS-2 and Cray T3D are network DMA machines. They are not, from the programmers perspective, truly shared memory, since transfers to "remote" memory must be done through special system calls. On the other hand, there is no synchronization or polling required to move data. Consequently `Get()` and `Put()` can be written as calls to these underlying vendor-supplied transport functions.

For all machines, a polling loop or interrupt must be used to detect a remote service request. Currently, Tulip uses a polling loop to detect requests. However, as active message layers for machines such as the SP2 become available, Tulip will be rewritten to take advantage of fast handlers and eliminate the need for polling.

2.3.3 I/O

pC++/streams is a library which supports a simple set of high level I/O primitives on pC++ collections. To illustrate its capabilities, we describe how pC++/streams can be used to checkpoint a collection having variable-sized elements.

Assume our application simulates the behavior of particles in three-dimensional space. We can model the particles with a one-dimensional distributed array of variable-length particle lists, each of which keeps track of the particles in the region of the three-dimensional array local to that processor.

```
class Position {
  double x, y, z;
};
```

```
class ParticleList {
  int numberOfParticles;
  double * mass;          // variable sized
  Position * position;    // arrays
 };

Collection DistributedArray {
  updateParticles();
};

Processors P;
Align a(12,"[ALIGN(collection[i], template[i])]");
Distribution d(12, &P, CYCLIC);

DistributedArray<ParticleList> particleArray(&d,&a);
```

The programmer can write a function to checkpoint the `particleArray` collection as follows:

```
#include "pc++streams.h"
void saveParticleArray() {
  oStream stream(&d, &a, "myFileOne");
  stream << particleArray;
  stream.write();
}
```

The first line of `saveParticleArray()` defines an output pC++/stream called `stream`, connected to the file `myFileOne`. The second line inserts the entire `particleArray` collection into the buffers of the stream. The third line causes those buffers to be written to the file, using parallel I/O. The file associated with the stream is closed automatically when the program block in which the stream was declared is exited. The programmer would write a function to restore the checkpointed `particleArray` as follows:

```
void loadParticleArray() {
  iStream stream(&d, &a, "myFileOne");
  stream.read();
  stream >> particleArray;
}
```

pC++/streams also allows selective I/O on individual fields of collection elements:

```
stream << particleArray.numberOfParticles;
```

pC++/streams supports I/O on collections with complex elements (e.g. variable-sized elements, tree-structured elements, etc) by giving the programmer a straightforward mechanism for defining how these data structures are to be read and written: stream insertion and extraction functions. A pC++/stream is actually a collection of element-streams, one per element of the collection to be written from or read into. An insertion or extraction function allows the programmer to indicate exactly how data is to be exchanged between a given element-stream and its corresponding element. In our example, the programmer would define an insertion function for `ParticleLists` as follows:

```
declareStreamInserter(ParticleList &p) {
  eltBuf << p.numberOfParticles;
  eltBuf << array(p.mass, p.numberOfParticles);
  eltBuf << array(p.position, p.numberOfParticles);
}
```

`declareStreamInserter()` is a macro that defines `eltBuf`, a reference to the element-stream. The `array()` macro tells pC++/streams that `mass` and `position` are dynamically-allocated arrays of size `numberOfParticles`. Extraction functions are defined similarly. pC++/streams is described in more detail in [16].

2.3.4 Persistence

pC++/persistence is an I/O library supporting persistence for pC++ collections. This library is currently implemented using the SHORE persistent object system from the University of Wisconsin-Madison [6].

Normally, elements of pC++ collections are transitory, i.e., their data disappears when the program terminates. In order to preserve transitory data, the programmer must output that data to a file before the program terminates, using either an I/O mechanism supported by the operating system or a higher-level library such as pC++/streams.

pC++/persistence allows programmers to define persistent collections, whose elements can contain persistent data in addition to ordinary transitory data. The persistent section of each element is automatically preserved across program executions; no application I/O code is required to save or load this data. A transaction mechanism is supported, allowing programmers to checkpoint persistent data with a single line of code that commits a transaction. In addition, the persistent part of a collection is concurrently accessible by multiple pC++ programs, with no explicit code for communication required. Concurrent access to persistent data can allow simplified programming of concurrent computation and visualization, computational steering, and modular multi-disciplinary simulations, since no application code needs to be devoted to I/O or communication of the persistent data.

As an example, we first we define the per-element persistent data using SDL (SHORE Data Language). For simplicity, our persistent data will consist of just a single long integer per element, called `myPersistentLong`:

```
module MyElement {
  interface PersistentElementData {
    public:
      attribute long myPersistentLong;
  };
}
```

This SDL specification is processed by the SHORE SDL type compiler, informing SHORE of the structure of the persistent part of our elements.

We next define an element class MyElement in ordinary pC++. We derive it from the class PersistentElement, and define an ordinary transient data member (myTransientLong) in the usual way:

```
#include "PersistentElement.h"

class MyElement : public PersistentElement{
 public:
  long myTransientLong;
  void P_initialize();
  void hello();
};
```

The class PersistentElement contains a member P through which the persistent part of each element is accessed:

```
void MyElement::hello() {
  printf(" Hello world: %ld %ld",
         myTransientLong, P->myPersistentLong);
}
```

The function P_initialize(), defined within MyElement, gives the application programmer a mechanism for initializing the persistent part of each element. P_initialize() is called immediately after the persistent part of each element is first created.

```
void MyElement::P_initialize() {
  P.update()->myPersistentLong = 1234;
}
```

The call to P.update() above informs pC++/persistence that the persistent part of the element is to be modified, rather than just accessed.

A persistent collection is defined just like an ordinary collection, except that it is derived from PersistentCollection:

```
#include "PersistentCollection.h"

Collection MyCollection: public PersistentCollection {
 public:
  MyCollection(Distribution *T, Align *A,
               char *persistentCollectionName);
 MethodOfElement:
  virtual void hello();
};
```

```
MyCollection::MyCollection(Distribution *T, Align *A,
                            char *persistentCollectionName)
  : PersistentCollection(T, A, persistentCollectionName) {}
```

When the programmer instantiates the collection X below, the string myPersistentCollectionName is passed into the collection constructor, and then to PersistentCollection. This string identifies a particular database of persistent elements to be associated with the collection.

```
void Processor_Main(int argc, char **argv){
  Processors P;
  Distribution T(SIZE, &P, BLOCK);
  Align A(SIZE,"[ALIGN(V[i], T[i])]");
  MyCollection<MyElement> X(&T, &A, "/myPersistentCollectionName");

  beginTransaction();
  X.hello();
  commitTransaction();
}
```

Changes to the persistent part of a collection must be made within a transaction, initiated by beginTransaction(). These changes do not become permanent and are not visible to other applications until the transaction is committed with a call to commitTransaction(). So to checkpoint the persistent part of a collection, all that is required is a call to commitTransaction().

pC++/persistence is still under development at the time of the writing of this text; some details may change and some functionality may be added before the implementation is complete.

2.4 An Example: Parallel Sorting

To see how pC++ is used, consider the problem of sorting a large vector of data using a parallel bitonic sort algorithm. A bitonic sequence consists of two monotonic sequences that have been concatenated together where a wrap-around of one sequence is allowed. That is, it is a sequence:

$$a_0, a_1, a_2, \ldots, a_m$$

where $m = 2^n - 1$ for some n, and for index positions i and j, with $i < j$, $a_i, a_{i+1}, \ldots, a_j$ is monotonic and the remaining sequence starting at $a_{(j+1) \bmod n}$, where a_0 follows a_n, is monotonic in the reverse direction.

Merging a bitonic sequence of length k involves a sequence of data exchanges between elements that are $k/2$ apart, followed by data exchanges between elements that are $k/4$ apart, etc. The full sort is nothing more than a sequence of bitonic merges. We start by observing that a set of two items is always bitonic. Hence for each even i, the subsequence a_i and a_{i+1} is always bitonic. If we merge

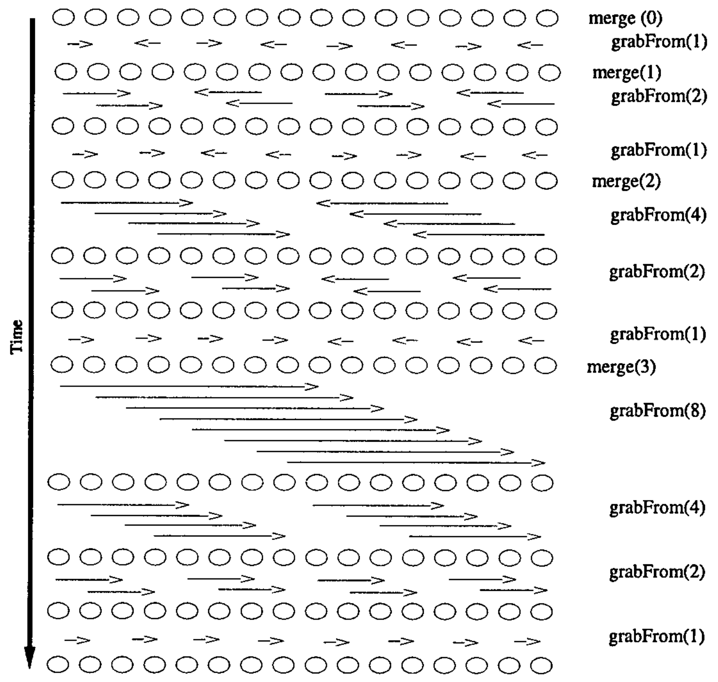


Figure 1: Data exchanges in the Bitonic Sort Algorithm

these length two bitonic sequences into sorted sequences of length two and if we alternate the sort direction, we then have bitonic sequences of length four. Merging two of these bitonic sequences (of alternating direction) of length 4 we have sorted sequences of length 8. The sequence of data exchanges is illustrated in Figure 1.

In pC++, a pure data-parallel version of this algorithm can be built from a collection `List` of objects of type `Item` as shown below. Each item contains an object of type `E` which is assumed to be the base type of the list we want to sort:

```
struct E {
 public:
  int key;
};
class Item {
 public:
  E a;
};
```

The `List` collection contains one public function `sort()` and a number of fields and members that are defined in the `MethodOfElement` section. Because the parallel algorithms require parallel data exchanges, we must have a temporary tmp to hold a copy of the data to be exchanged for each element. In addition, there are two flags, exchangeDirection and sortDirection which are used to store the current exchange direction and the current sort order respectively. As can be seen in figure Figure 1, the value of these flags depends on the location of the element in the list as well as the point in time when an exchange is made.

The pC++ definition of the List collection can be summarized as follows.

```
Collection List: SuperKernel {
 public:
  void sort();
  int N;  // number of elements
 MethodOfElement:
  E tmp;
  virtual E a;
  int sortDirection, exchangeDirection;
  void set_sort_direction (int k) { sortDirection  = (index1/k)%2; }
  void set_exchange_direction(int k) { exchangeDirection = (index1/k)%2; }
  void merge(){
    if (((sortDirection == exchangeDirection) && (this->a.key > tmp.key)) ||
        ((sortDirection != exchangeDirection) && (this->a.key <= tmp.key))){
      this->a = tmp;
    }
  }
  void grabFrom(j){
    if(exchangeDirection == 1) tmp = (*thisCollection)(index1+j)->a;
    else        tmp = (*thisCollection)(index1-j)->a;
  }
};
```

In general, MethodOfElement functions are those element-wise operations in an algorithm that depend on the relation of one element to the whole collection or to other elements in the collection. For example, the function grabFrom(int j) is a method that, when applied to one element at position k, will access the data in the element at position j+k.

The SuperKernel class provides two additional members, thisCollection and index1, which provide a pointer to the containing collection and the position of the element in the collection, respectively. The function merge() uses the current state variables sortDirection and exchangeDirection to determine which element of the data to keep after the exchange step.

The sort() function is then a sequence of merge steps, each of which contains a sequence of exchanges as shown below. The main program allocates a list of items and then calls the sort function.

```
List::sort(){
  int k = 1;
  for (int i = 1; i < log2(N); i++){  // merge(i) step
    k = 2*k;
    this->set_sort_direction(k);
    for (int j = k/2; j > 0; j = j/2){ // exchange(j) step
      this->set_exchange_direction(j);
      this->grabFrom(j);
      this->merge();
    }
  }
}

Processor_main(){
  Processors P;
  int N = read_problem_size();

  Distribution D(N,&P,BLOCK);
  Align A(N,"[ALIGN(X[i],D[i])]");
  List< Item >  L(&D, &A);
  ...
  L.sort();
}
```

This version of the program works, but has a serious flaw. If the size of the list to be sorted is N and there are only $P << N$ processors in the system, the bitonic sort has parallel complexity $O(\frac{N}{P}\log^2 N)$, which is far from optimal. To improve the efficiency, we can build a hybrid algorithm as follows. Let us break the list of N into P segments of length $K = \frac{N}{P}$. We begin the sort by applying a quicksort to each segment, but sorting them in alternating directions. Now each pair of adjacent sorted segments forms a bitonic sequence and we can apply the bitonic merge as before. However, at the end of each merge step, the list in each segment is only a bitonic sequence, not a sorted sequence. We must then apply a local bitonic merge to sort it. If we rewrite the algorithm above with a `Segment` class replacing the `Item` class and expanding the `tmp` variable to an array we only need to make a few modifications to the program. These changes, shown below, consist of inserting the calls to the local quicksort and local bitonic merge in the sort function. The `grabFrom()` and `merge()` functions also need to be replaced by ones that can accommodate an array.

```
// P is the number of elements (processors)
// N is the total number of elements to sort.
// K = N/P is the size of each segment.

class Segment{
 public:
  E a[K]
  quickSort(); // O(K log(K))
  localBitonicMerge(int direction); // O(K)
};

List::sort(){
  int k = 1;
  this->quickSort();
  for (int i = 1; i < log2(P); i++){  // merge(i) step
    k = 2*k;
    this->set_sort_direction(k);
    for (int j = k/2; j > 0; j = j/2){ // exchange(j) step
      this->set_exchange_direction(j);
      this->grabFrom(j);
      this->merge();
    }
    this->localBitonicMerge(d);
  }
}

void SortedList::grabFrom(int dist){
  E *T;
  int offset = (d2)? -dist: dist;
  T = &((*ThisCollection)(index1+offset)->a[0]);
  for(int i = 0; i < K; i++) tmp[i] = T[i];
}
void SortedList::merge(){
  for (i = 0; i < K; i++)
    if(((d == d2) && (a[i].key > tmp[i].key)) ||
       ((d != d2) && (a[i].key <=tmp[i].key))){
      a[i] = tmp[i];
    }
  }
};
```

Assume that the quicksort computation runs with an average execution time of $DK \log K$ for some constant D, that we can ignore the cost of the barrier synchronization, that there are $P = 2^n$ processors available, and that the size of the list to be sorted is N. The time to sort is then roughly:

$$T(N) = \frac{N}{P} C \log^2 P + D \frac{N}{P} \log \frac{N}{P} + \log P$$

where C is a constant that depends upon communication speed. Given a sequential complexity of $DN \log N$ we see that the parallel speed-up is of the form:

$$Speedup(N, P) = \frac{P}{1 + \frac{C \log^2 P}{D \log N}}$$

which, for large N, approaches P.

This algorithm has been tested on a variety of machines and it is both portable and fast. Sorting one million items takes 3.56 seconds on a 64 node Paragon and 1.68 seconds on an 8 node SGI Challenge. However, comparing this to the standard system routine qsort reveals that the speedup is not great. On the same data set with one node of the SGI Challenge, qsort requires 10.21 seconds. Hence the speed-up of our algorithm is 6.08 on 8 processors. This value matches the formula above when $C = D$.

2.5 The Polygon Overlay Program

The following algorithm is used to implement the polygon overlay code in pC++. Given two maps A and B as input, map A is divided into smaller maps A_s. These smaller maps are then distributed over the elements of a pC++ collection. If there are N polygons in map A to be divided and P collection elements, then each element gets N/P polygons, except element zero, which gets $N/P + N \bmod P$ polygons (Figure 2).

Map B is duplicated in each element. During a parallel computation, each element finds the overlay of map A_s and map B. In the output stage, the resulting overlay map in each element is combined with the maps in the other elements to form the final overlay map. No inter-element communication is required during the parallel computation and thus the computation is carried out in the "embarrassingly parallel" fashion. In this algorithm, map B is not divided and distributed; if it were, it would be difficult for an A_s map to know whether it overlaps with a B_s map which is in another element. A more elaborate parallel algorithm would be required, and inter-element communication would be unavoidable. We discuss this further later in this section.

The collection pC++ element class is defined as follows:

```
class Patch {
 public:
  polyVec_p leftVec, rightVec, outVec;
  Patch() {}
};
```

where leftVec, rightVec, and outVec are, respectively, pointers to map A_s, map B, and their resulting overlay. The pC++ collection is defined as follows:

```
Collection Overlay : public SuperKernel {
 public:
  Overlay(Distribution *D, Align *A);
 MethodOfElement:
  virtual polyVec_p leftVec, rightVec, outVec;
  void readMap();
  void writeMap();
  void distributeMap();
  void findOverlay();
};
```

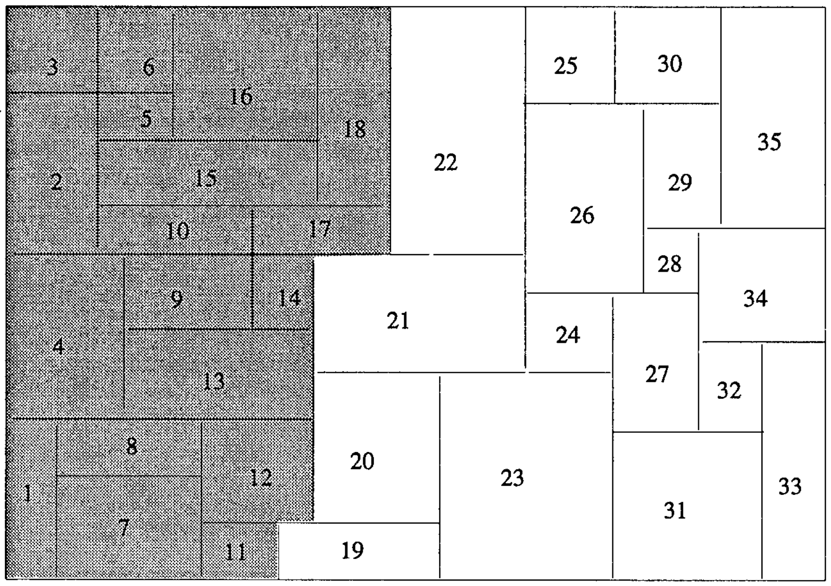


Figure 2: Polygon map distribution scheme. This shows the distribution of a map consisting of $N = 35$ polygons over $P = 2$ collection elements. The polygons are numbered (sorted) according to the order used in the sequential ANSI C implementation. With $N/P = 17$ and $N \bmod P = 1$, element 0 gets the shaded polygons and element 1 gets the unshaded polygons. In the tests reported in this paper, the load imbalance was found to be insignificant.

In this definition, readMap() inputs the two polygon maps. The actual reading is carried out by collection element zero. After the two maps are read, element zero calculates the number of polygons all other elements should have and broadcasts the information. In distributeMap(), all other processors then fetch their piece of the first polygon map and the entire second polygon map from element zero. findOverlay() finds the overlay of A_s and B maps. In its pC++ implementation, findOverlay() simply calls the original ANSI C polygon overlay functions based on user-selected options. No modification of the ANSI C code is needed, except in the case of a modified list-deletion algorithm described in later in this section.

In writeMap(), element zero gathers overlay maps from all the elements. It calls a sorting routine to sort the polygons in a special order and writes the entire overlay map out. The sorting could have be done in parallel using the modified bitonic merge sort described in (Section 2.7.1). However, since our focus was on the parallelization of the polygon overlay algorithm itself, we did not parallelize the sorting routine. The actual implementation of function findOverlay() is given in the following piece of code.

```
void Overlay::findOverlay()
{
  double time;
  pcxx_UserTimerClear(index1);
  pcxx_UserTimerStart(index1);
  if (useLnArea && useOrder){
    /* sorted-ordered list-deletion overlay */
    outVec = overlayAreaLinkedOrder(leftVec, rightVec);
  } else ···{
    ···as in sequential code···
  }
  pcxx_UserTimerStop(index1);
  time = pcxx_UserTimerElapsed(index1);
  printf("Time for element %d : %lf", index1, time);
}
```

where pcxx_UserTimer functions clear, start, and stop a timer numbered by the element's index. pcxx_UserTimerElapsed reports the elapsed time. The main program is:

```
void Processor_Main() {
  int elem_count = pcxx_TotalNodes();
  Processors P;
  Distribution D(elem_count,&P,BLOCK);
  Align A(elem_count,"[ALIGN(X[i],D[i])]");
  Overlay<Patch> X(&D,&A);
  X.readMap();
  X.distributeMap();
  X.findOverlay();
  X.writeMap();
```

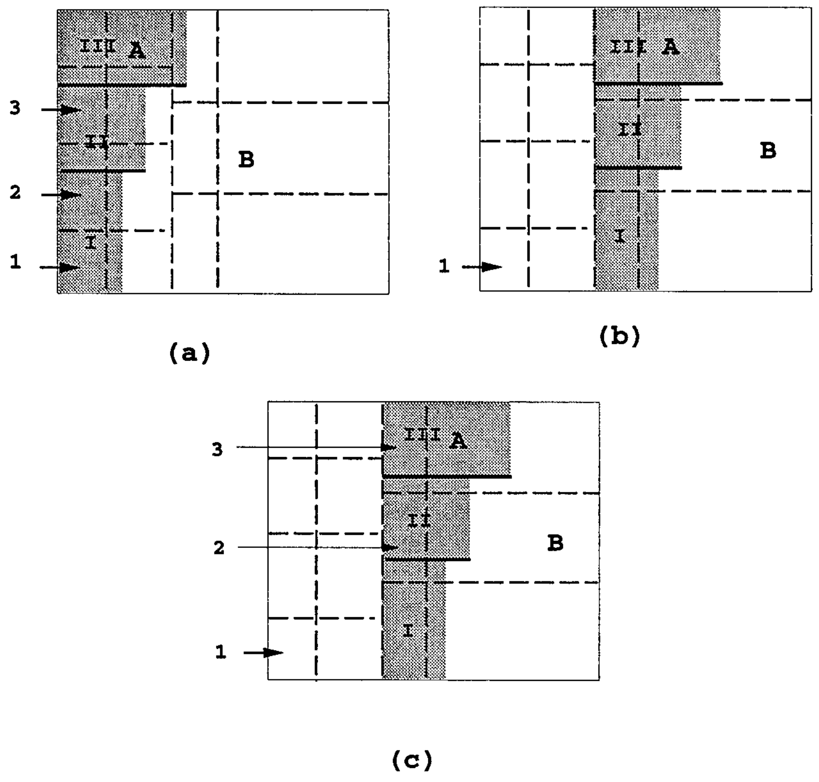


Figure 3: Comparing the list-deletion with the modified list-deletion algorithm.

```
}
```

where pcxx_TotalNodes returns the number of processors used for the computation.

The pC++ code was tested on a variety of platforms including a Cray T3D, an IBM SP-2, a SGI Power Challenge, an Intel Paragon and a Sun Sparc 10. Two maps each containing about 60,000 polygons were used as input. Three sets of tests were conducted using the naive overlay algorithm, the list-deletion overlay algorithm, and a modified list-deletion overlay algorithm.

The modified list-deletion algorithm can be described as follows. As illustrated in figure 3, we are given two maps A and B. A is indicated by shaded area. Polygons in map A are separated by solid lines. Polygons in B are separated by dashed lines. Assume the polygons are sorted according to the x coordinates of their upper right corner, the ordering scheme used in the ANSI C code, so that comparison of the two maps would begin with the polygons in the lower left corners of both maps. In both the modified and the original list-deletion algorithms, when the lower left corners of maps A and B coincide, the loop which compares polygons in B with polygon I in A begins with the polygon pointed by arrow 1. The subsequent comparisons of polygons in B with polygon II in A begin with the polygon pointed arrow 2, because the polygon pointed by arrow 1 has been eliminated in earlier comparisons. Similarly, subsequent comparisons of polygons in B with polygon III in A begin with the polygon pointed arrow 3. In list-deletion algorithm, when the lower left corners of the maps do not coincide, all comparisons of polygons in B with polygons I, II, III in A begin with the polygon pointed by arrow 1. This is because polygons to the left of map A are never eliminated in the comparison process. In the modified list-deletion algorithm, when the lower left corners of the maps do not coincide, only comparisons involving polygon I begin with the polygon pointed by arrow 1. Subsequent comparisons involving II and III begin with the polygons pointed by arrow 2 and 3 respectively.

Our experiments with the list deletion algorithm revealed that it does not scale well. The extra work required to compare polygons in map A with polygons in map B where no overlay occurs can degrade the algorithms performance to well below that of the naive algorithm. Because the polygons in all the maps we used for our tests were already sorted, the modified the list-deletion overlay algorithm could be applied. The resulting improvement in performance was dramatic. The benchmark results of the three sets of experiments are shown in Table 2 and Figure 4.

As can be seen in Table 2 and Figure 4, on all the machines we were able to obtain nearly linear speedups for the naive and the modified list-deletion algorithm. The speedup curves decreased slightly for the modified list-deletion algorithm as the number of processors increased. This was due to the fact that as workload on each processor decreased, the overhead became more prominent. The results show that the parallel algorithm we adopted worked very well for the naive and the modified list-deletion algorithm. The original list-deletion algorithm is not well-suited for parallel execution, causing the parallelized code to perform poorly.

Another way to parallelize the list-deletion algorithm without modifying the sequential list-deletion algorithm is to divide B into B_s and distribute B_s as we did with A_s. Assuming, after the division and distribution, A_s and B_s roughly cover the same area, finding the overlay of them will be straight forward. Once the overlay of A_s and B_s is found, the collection elements exchange the part of B_s where no overly is found, and a second phase of parallel operation can be carried out. This parallel algorithm requires N phases of parallel operation where N is the number of collection elements (usually chosen to be equal to the number of processors). This algorithm also requires that the input polygons be sorted.

However, it should be noted that the result of the modified list-deletion algorithm is a distributed list of polygons which are locally sorted but not globally sorted. However globally sorting the polygons is a very simple task. The sorting algorithm described in section Section 2.4 has been applied to a data set of

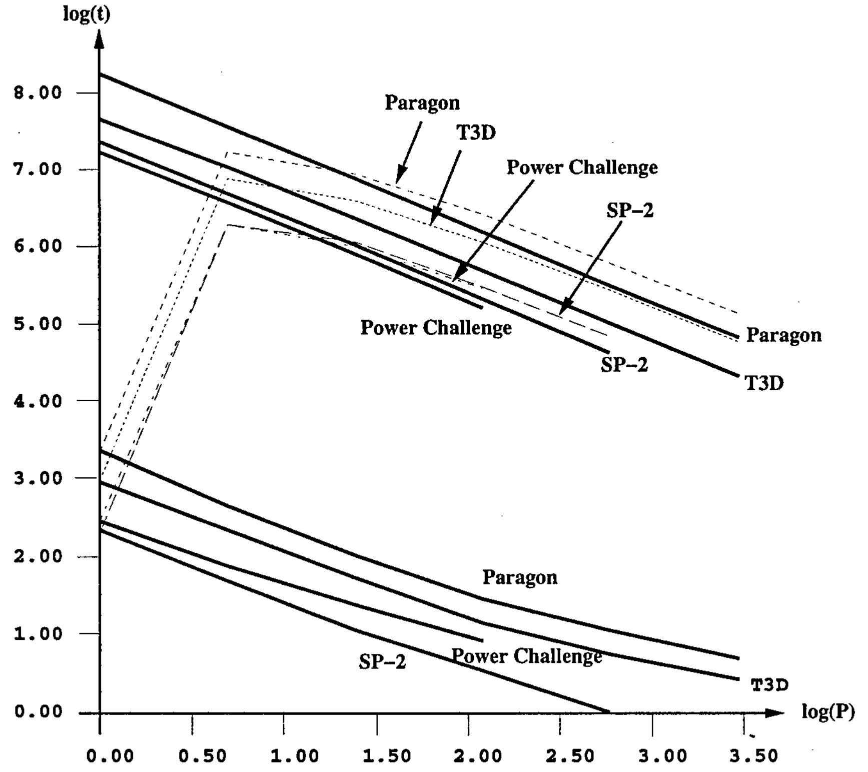


Figure 4: log(P) vs. log(t) plot. P is the number of processors and t is the execution time in seconds. The upper four solid lines are log(P) vs. log(t) curves for the naive overlay algorithm; the lower four solid lines are log(P) - log(t) curves for the modified list-deletion algorithm; the four dashed lines are the log(P) vs. log(t) curves for the list-deletion algorithm.

Platform	Number of Processors						
	1	2	4	8	16	32	64
Cray T3D	2135.7 19.9 19.5	1143.2 990.5 10.6	590.2 735.7 5.7	299.8 429.6 3.2	151.1 229.6 2.1	75.8 118.2 1.5	
Intel Paragon	3782.4 28.5 29.1	1942.4 1414.8 14.5	983.4 1048.7 7.6	494.9 612.7 4.3	248.1 327.6 2.9	124.2 168.8 2.0	 85.6 1.66
IBM SP-2	1587.7 10.2 10.6	812.1 554.4 5.4	410.4 430.1 2.9	205.7 238.8 1.7	103.3 127.6 1.0		
Power Challenge	1409.8 11.7 11.7	724.8 547.1 6.6	367.6 405.4 4.0	185.1 236.2 2.5			
SPARC 10	1562.3 14.0 13.4						

Table 2: Time, in seconds, spent in the `findOverlay` function. Two maps each containing about 60,000 polygons (file K100.00 and K100.01) were used as input. For each platform, results are shown for the naive overlay algorithm first, then for the list-deletion overlay algorithm, then for the modified list-deletion overlay algorithm.

this size and the time to sort it was 0.35 seconds on an 8 processor SGI Power Challenge. Hence the execution times in the table above should have about one third of a second added to account for the final sort.

A large fraction of the code in many parallel applications is devoted to I/O. For example, in an early version of the polygon overlay program using ordinary UNIX file I/O, 200 lines of code (approximately 10% of the total), was devoted to I/O. In addition to programming time overhead for file I/O, there is run time overhead as well; I/O is increasingly being identified as a bottleneck in parallel applications.

pC++/streams (Section 2.3.3) can reduce the programming time and run time overheads associated with file I/O in pC++ applications. Rewriting the original UNIX I/O in the polygon overlay program using pf-streams reduced I/O code from 200 lines to 70 lines.

2.6 The Self-Consistent Field Code

Here and in Section 2.7 we describe our work with with the Grand Challenge Cosmology Consortium (GC^3). This work is abstracted from two longer papers: [32] and [15].

One of the N-body codes developed by the GC^3 researchers is the Self-Consistent Field (SCF) code, which is used to simulate the evolution of galaxies. It solves the coupled Vlasov and Poisson equation for collisionless stellar systems using the N-body approximation approach. To solve Poisson's equation for gravitational potential:

$$\nabla^2 \Phi(\vec{r}) = 4\pi\rho(\vec{r}),$$

the density ρ and potential Φ are expanded in a set of basis functions. The basis set is constructed so that the lowest order members well-approximate a galaxy obeying the de Vaucouleurs $R^{1/4}$ projected density profile law. The algorithm used is described in detail in [17].

The original SCF code was written in Fortran-77 by Lars Hernquist in 1991. In 1993, the code was converted to Thinking Machines CM Fortran by Greg Bryan. Experiments conducted by Bryan on the 512-node CM-5 at the National Center for Supercomputing Applications (NCSA) indicate that with 10 million particles the CMF code can achieve 14.4 Gflops on 512 nodes of the CM-5 [17].

The expansions of the density and potential take the following forms:

$$\rho(\vec{r}) = \sum_{nlm} A_{nlm} \rho_{nlm}(\vec{r})$$

$$\Phi(\vec{r}) = \sum_{nlm} A_{nlm} \Phi_{nlm}(\vec{r})$$

where n is the radial quantum number and l and m are quantum numbers for the angular variables. Generally, the two sums will involve different expansion coefficients. But the assumption of bi-orthogonality ensures a one-to-one relationship between terms in the expansions for the density and potential. The basis sets ρ_{nlm} and Φ_{nlm} also satisfy Poisson's equation:

$$\nabla^2 \Phi_{nlm}(\vec{r}) = 4\pi\rho_{nlm}(\vec{r})$$

and are given by:

$$\rho_{nlm}(\vec{r}) = \frac{K_{nl}}{2\pi} \frac{r^l}{r(1+r)^{2l+3}} C_n^{2l+3/2}(\xi)\sqrt{4\pi} Y_{lm}(\theta, \phi)$$

$$\Phi_{nlm}(\vec{r}) = -\frac{r^l}{(1+r)^{2l+1}} C_n^{2l+3/2}(\xi)\sqrt{4\pi} Y_{lm}(\theta, \phi)$$

$$\xi = \frac{r-1}{r+1}$$

where K_{nl} is a number related only to n and l, and $C_n^{2l+3/2}(\xi)$ and $Y_{lm}(\theta, \phi)$ are ultraspherical polynomials and spherical harmonics, respectively. After some algebra, the expansion coefficients become

$$A_{nlm} = \frac{1}{I_{nl}} \sum_k m_k [\Phi_{nlm}(r_k, \theta_k, \phi_k)]^*$$

where I_{nl} is a number and m_k is the mass of the kth particle. Once the gravitational potential is found, the gravitational force per unit mass can be obtained by taking the gradient of the potential and the particles can be accelerated accordingly.

2.6.1 The pC++ Version of the SCF Code

We design a C++ class called `Segment` to represent a subgroup of the N particles used in the simulation. As we have discussed earlier, the major procedure in the SCF code is to compute the sums for the expansion coefficients A_{nlm}. Our approach is to first compute local sums within each `Segment` object. After this, global sums are formed by a global reduction. The global sums are then broadcast back to each `Segment` object where the particles are accelerated by the gravitational force. Fortran subroutines in the original Fortran code can be used as member functions of the `Segment` class, although subroutines involving inter-element communication and I/O need to be modified.

The Fortran subroutines are called by pC++ through a specially designed Fortran interface [32]. The `Segment` class is declared (with many unimportant variables and member functions omitted) as follows:

```
class Segment {
 public:
  FArrayDouble x, y, z, vx, vy, vz, ax, ay, az, mass,
               plm, clm, dlm, elm, flm, dplm;
  double sinsum[lmax+1][lmax+1][nmax+1],
         cossum[lmax+1][lmax+1][nmax+1];
  Segment();
  void compute_polynomial();
  void compute_acceleration();
  void update_position();
  void update_velocity();
};
```

The data type `FArrayDouble` is defined in the Fortran library; it serves as an interface to Fortran double precision arrays. The `FArrayDobule` variables defined above are one-dimensional arrays that contain the positions, the velocities, the accelerations, and the masses of particles belonging to a `Segment` object, and the expansion coefficients and values of the polynomial. `sinsum` and `cossum` contain the local sums and eventually the global sums of the expansion coefficients. The class member functions call Fortran subroutines: compute_polynomial() computes the polynomials and local sums, compute_acceleration() computes the acceleration for each particle, and update_position() and update_velocity() update the positions and velocities of particles.

The collection that distributes the elements, allocates memory, and manages inter-element communication is declared as below. Again, many less important member functions are omitted for brevity:

```
Collection SelfConsistField : public Fortran {
 public:
  SelfConsistField(Distribution *D, Align *A);
  void InParticles();
  void InParameters();
  void OutParticles(int nsnap);
 MethodOfElement:
  virtual void compute_polynomial();
  virtual void compute_acceleration();
  virtual void update_position();
  virtual void update_velocity();
  void read_segment();
  void write_segment();
};
```

The functions declared here are pC++ functions. Their main purpose is to handle I/O. InParticles(), InParameters(), and OutParticles() read input files and write to output files, while read_segment() and write_segment() are called by InParticles() and OutParticles() to perform parallel I/O. Functions that are already defined in element class Segment are declared as virtual functions in this collection declaration. The inherited Fortran collection is a parent collection which handles inter-element communication. Fortran itself is derived from the SuperKernel collection.

The main program is:

```
void Processor_Main() {
  elem_count = pcxx_TotalNodes();
  Processors P;
  Distribution D(elem_count, &P, BLOCK);
  Align A(elem_count, "[ALIGN(X[i], D[i])]");
  SelfConsistField<Segment> X(&D, &A);
  // read initial model
  X.InParameters();
  X.InParticles();
  X.compute_polynomial();
  X.ReduceDoubleAdd(offset,variable_count);
  X.compute_acceleration();
  // main loop
  for (n = 1; n <= nsteps; n++) {
    X.update_position();
```

Platform	Number of Processors			
	8	16	32	64
Cray T3D		223.0	115.3	
Intel Paragon		667.3	332.5	168.5
IBM SP-2		186.9	103.5	
Power Challenge	116.9	58.6		
CM-5 (pC++)			45.8	
CM-5 (CM Fortran)			50.3	

Table 3: SCF code execution time, in seconds, for evolving a 51,200 particle stellar system for 100 time steps. The expansions were truncated at $nmax = 6$ and $lmax = 4$.

```
    X.compute_polynomial();
    X.ReduceDoubleAdd(offset, variable_count);
    X.compute_acceleration();
    X.update_velocity();
    X.OutParticles(n);
  }
}
```

where ReduceDoubleAdd is a reduction function inherited from SuperKernel. offset is measured from the beginning of the class Segment to the beginning of the field sinsum, and variable_count is the total number of array elements in sinsum and cossum. A leapfrog integration scheme is used to advance particles.

2.6.2 Benchmark Results

Our experiments with the pC++ SCF code were conducted on a Thinking Machines CM-5, an Intel Paragon, an SGI Power Challenge, an IBM SP-2, and a Cray T3D. For comparison, we also ran the CM Fortran SCF code on the CM-5. 51,200 particles were used for the simulation. The system was allowed to evolve for 100 time steps. The results of these experiments are listed in Table 3

As can be seen, the SCF code scales up very well on the parallel machines. On the CM-5 the pC++ version is about 1.1 times faster than the CM Fortran code. This is mainly because the pC++ code used a faster vector reduction routine, while the CM Fortran code used a scalar reduction routine. The code achieved highest speed—approximately 50 MFLOPS per processor—on the SGI Power Challenge. [3]

[3]The pC++ version of the SCF code described here was used recently in an experiment involving a simulation of 16 million particles. One of the largest such simulations to date. The computation was distributed over two MPPs, the 512 node NCSA CM-5 and the 512

2.7 The Particle Mesh Code

Another N-body code in the dossier of the GC^3 group is the Particle Mesh (PM) code [9]. Originally implemented in Fortran-77 and CM Fortran, the particle-mesh method used in the PM code computes long-range gravitational forces in a galaxy or galaxy cluster system by solving the gravitational potential on a mesh. The three-dimensional space is discretize by a three-dimensional grid. An average density for each grid point is then computed using a Nearest Grid Point scheme, in which the density value at a grid point is the sum of all masses of the particles nearest to that grid point. Once the density values at the grid points are known, Fourier transforms are performed to compute the potential values at those points. The potential values at the grid points are finally interpolated back to the particles, and the particle positions and velocities are updated. The natural data structures for this are an one-dimensional particle list and a three-dimensional mesh.

2.7.1 The Particle List Collection

The particles in the simulation are first sorted according to their affinity to mesh points; particles closest to a given mesh point are neighbors in the sorted list. The sorted list is then divided into segments and each segment forms an element of a particle list collection.

There are two approaches that we can follow when dividing the sorted list. There is a tradeoff between data locality and load balance associated with the two approaches. In the first approach, the sorted list is evenly divided so that the segments have the same length. In the second, particles belonging to the same mesh points are grouped into the same segment and segments will have different lengths.

In the first approach, load balancing is ensured because each processor has the same number of particles. However, this approach may cause particles belonging to the same mesh point to be distributed among different elements, thus requiring more inter-element communication and remote updates. The second approach allows a greater exploitation of data locality, but there is a potential load balancing problem. As the system evolves, particles (stars or galaxies) tend to group together into clumps. Consequently, some mesh points may have 1000 times more particles than other mesh points, and segments that have these mesh points will have much longer lengths. Since we usually distribute the collection elements (in this case segments) evenly across the processors in a parallel machine, the processors that have those long segments will do more work. We therefore decided to follow the first approach.

The `Segment` class is defined as:

```
class Segment {
```

node PSC T3D and run in parallel. Communication bettween the two codes was done using MPICH over the internet.

```
 public:
  int particle_count;
  FArrayDouble x, mass, g, v;
  Segment();
};
```

where x, mass, g, and v represent the position, the mass, the acceleration induced by gravity, and the velocity of a particle, respectively.

The ParticleList collection is defined as

```
Collection ParticleList : public Fortran {
 public:
  ParticleList(Distribution *D, Align *A);
  void SortParticles();
 MethodOfElement:
  void pushParticles(Mesh<MeshElement> &G);
  void updateGridMass(Mesh<MeshElement> &G);
  ...
};
```

The function SortParticles() sorts particles in lexicographic order according to their positions. The particles within each segment are first sorted using the standard C library quicksort function qsort(). A global parallel sort is then performed using the bitonic sort of Section 2.4.

pushParticles() uses the gravitational force to update the positions and velocities of the particles. The argument passed to pushParticles() is a collection designed for the mesh data structure (see next subection). The mesh collection is passed to pushParticles() so that potential values at the grid points can be accessed by particles in the Segment element and used to update the particles' velocities and positions. The function updateGridMass() is used to add the mass of a particle to the total mass of the mesh point to which it is closest. This function first loops through the particles local to a segment and accumulates a local total mass for each mesh point. It then adds the local total mass to the mesh point's total mass by a remote update operation on the appropriate mesh point. Because remote updates are expensive, the particles are sorted to minimize the number of remote updates.

2.7.2 The Mesh Collection

The mesh is logically a three dimensional array of mesh points, each containing values for density and position. Because an FFT is used to solve the gravitational potential equation, the data structure is designed as an one-dimensional collection, each element of which contains a slice of the three-dimensional mesh:

```
class MeshElement {
 public:
  double density[x_dim_size][y_dim_size];
  MeshElement();
  void add_density(double density, int x_zone, int y_zone);
};
```

add_density() is remotely invoked by Segment elements to deposit mass on grid points.

The collection Mesh is defined as:

```
Collection Mesh : Fortran {
 public:
  Mesh(Distribution *T, Align *A);
  void computePotential();
 MethodOfElement:
  void xyFFT_forward();
  void zFFT_forward();
  void zFFT_backward();
  void xyFFT_backward();
  void transpose_xy_to_xz();
  void transpose_xz_to_xy();
};
```

The function computePotential() computes the gravitational potential using the total mass at each mesh point. It calls the FFT routines listed under MethodOfElement. The density distribution is first transformed into the wavenumber domain by a FFT along the x, y, and z directions. After solving the Poisson's equation for the gravitational potential in the wavenumber domain, the potential (or force components) is transformed back into the spatial domain.

The FFT transform in the x, y, and z directions is performed by the Mesh collection. The FFT in the x and y directions is straightforward, since each MeshElement contains an entire array of mesh points. To perform an FFT in the z-direction, data are transformed using transpose_xy_to_xz and transpose_xy_to_xy.

2.7.3 The Main Simulation Loop

Given these collections, the main body of the simulation can be implemented as follows:

```
main(){
 int num_of_segments = pcxx_TotalNodes();
 int mesh_dim_z = 64;
 Processors P;

 Distribution Dist_PartList(num_of_segments, &P, BLOCK);
 Align Align_PartList(num_of_segments, "[ALIGN(G[i], T[i])]");
 ParticleList<Segment> part(&Dist_PartList, &Align_PartList);

 Distribution Dist_Mesh(mesh_dim_z, &P, BLOCK);
```

Platform	Number of Processors			
	8	16	32	64
Cray T3D		33.4	23.1	
IBM SP-2			81.0	
Power Challenge	30.4	16.1		
CM-5 (pC++)			134.6	
CM-5 (CM Fortran)			20.4	

Table 4: PM code execution time, in seconds, for evolving a 32,768 particle stellar system for 10 time steps. A $64 \times 64 \times 64$ grid was used.

```
 Align Align_Mesh(mesh_dim_z, "[ALIGN(G[i], T[i])]");
 Mesh<MeshPlane> mesh(&Dist_Mesh, &Align_Mesh);

 // initialize particle list
 ...
 // main loop
 for (int i = 0; i < number_of_steps; i++){
   mesh.computePotential();
   particlelist.pushParticles(mesh);
   particlelist.sortParticles();
   particlelist.updateGridMass(mesh);
 }
 ...
}
```

The main loop involves computation on both the Mesh and ParticleList collections. First, the potential is computed in parallel on the grid. Second, the particle velocities and positions are updated. If particles have moved to new grid points, the appropriate data structures are then updated. The particles are then sorted, after which the particle masses are accumulated in their corresponding points for the next iteration step.

2.7.4 Benchmark Results

Our experiments with the pC++ PM code were conducted on a Thinking Machines CM-5, an Intel Paragon, an SGI Power Challenge, an IBM SP-2, and a Cray T3D. For comparison, we also ran the CM Fortran PM code on the CM-5. 32,768 particles were used for the simulation. The system was allowed to evolve for 10 time steps. The results of these experiments are listed in Table 4.

As can be seen in the table, the code scales up relatively well on the T3D and Power Challenge. On the CM-5, the pC++ code is considerably slower than the CM Fortran code. This is because the CM Fortran code can make use of transpose routines embedded in an FFT developed by Thinking Machines' engineers. The pC++ code has complicated data structures and cannot use those transpose routines. Again, the best performance was obtained on the Power Challenge, although this architecture is limited to a small number of processors.

In the following chapter we describe the TAU performance analysis tool system that provides the programming and debugging environment for pC++. TAU is a complete system that provides pC++ with an *integrated development environment* similar to that found in professional development system. However, by providing parallel performance analysis capabilities, TAU goes far beyond any available commercial system.

3 Tau

3.1 Introduction

Most users find parallel programming difficult for at least four reasons. First, parallel computing abstractions (e.g., data parallelism, task parallelism, producer/consumer parallelism) are more diverse than sequential abstractions, and the range of abstractions is not as well supported in existing programming systems. ARE PARALLEL COMPUTING ABSTRACTIONS MORE DIVERSE THAN LIST-BASED VS. UNIFICATION-BASED VS. PROCEDURAL VS. ...? Hence, the choice of a parallel computing model requires a sophisticated understanding of application, algorithm, language, system, and architecture issues. WHAT ARE "SYSTEM" ISSUES (THAT ISN'T COVERED BY OTHER HEADINGS IN THIS LIST? 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 2). Our charter was to design and develop a program analysis tool system. CAN YOU SUPPORT THIS CLAIM TO UNIQUENESS? WHAT DOES IT MEAN TO SAY, "LET THE PROGRAMMING REQUIREMENTS DETERMINE THE TOOL SPECIFICATIONS"? The project was unique in that we decided early on to let the programming requirements determine the tool specifications. 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 TAU, for **T**uning and **A**nalysis **U**tilities. In Section 3.2, we list the requirements that we felt TAU had to address. How TAU's design meets these requirements is discussed in Section 3.3, which also covers TAU's architecture and implementation. Each TAU tool is described in full in Section 3.4. Since the effectiveness of TAU can be only measured by its usefulness for analysis of pC++ programs, Section 3.5 shows how TAU was used to analyze the pC++ implementation of polygon overlay, described in section 2.5.

TAU is not without its shortcomings. A critique of its current state is given in Section 3.6. Section 3.7 discusses future development of TAU, partiuclarly its extension to other parallel language environments and the incorporation of more sophisticated tools.

3.2 Design Requirements and Goals

An earlier version of this material was published in [24].

TAU 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 retarteable, 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).

3.3 TAU Overview

Parts of this material (3.0 and 3.1) were previously published in [24].

The TAU architecture defines how its components interoperate and fit in the pC++ language system. Below, we describe the TAU design and show how it addresses the programming productivity requirements of pC++. TAU 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.

TAU was specifically designed to meet the requirements listed in the previous section:

Give a user (program-level) view. Elements of the TAU graphical interface represent objects of the pC++ programming paradigm: collections, classes, methods, and functions. These language-level objects appear in all TAU utilities.

Support high-level, parallel programming languages. TAU 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 TAU were to determine what low-level instrumentatione 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. TAU uses the Sage++ toolkit [3] as an interface to the pC++ compiler for instrumentation and accessing properties of program objects. TAU is also integrated with the runtime system of pC++ for profiling and tracing support.

Enable portability, extensibility, and retargetability. We implemented TAU in C++ and C to ensure an efficient, portable, and reusable implementation. The same reasoning led us to use Tcl/Tk [26] for our graphical interface.

The TAU 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 TAU can be retargeted to other programming environments.

Enhance usability. We tried to make the TAU 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 TAU tools also support *global features*. When a global feature is invoked in any tool, it is automatically executed in all TAU tools which are currently running. Examples of these global features are described later. TAU also includes a full hypertext help system.

3.3.1 TAU Architecture

Figure 5 shows an overview of the pC++ programming environment. The pC++ compiler front end 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 3.4.6). The pC++ back end 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 11[4]). Through a graphical interface, the user first selects the parallel machine on which

[4]Figures 10 to 26 appear in Section 3.5

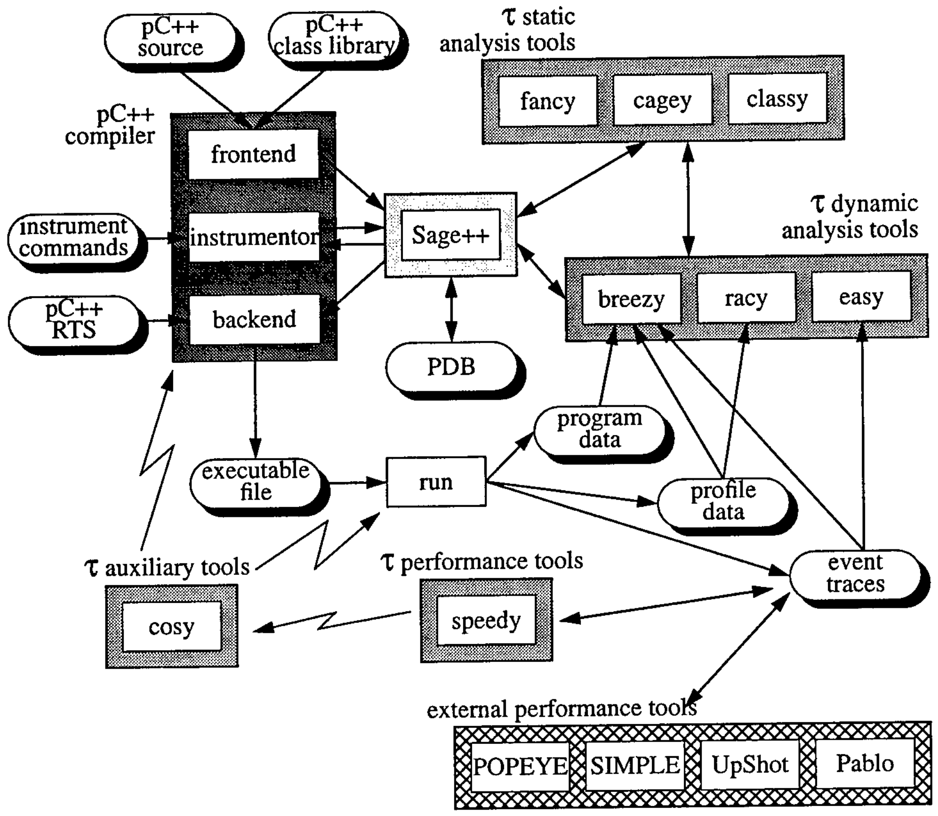


Figure 5: TAU Tools Architecture

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 5. It includes the TAU toolset, instrumentation, profiling, tracing, and breakpointing support, and interfaces to performance analysis tools developed by other groups [18, 23, 27]. The TAU tools are described in more detail in Section 3.4.

3.3.2 TAU Implementation

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

3.3.3 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), TAU allows users to click on anything which represents an function or class in any tool, and automatically updates all tool display to show information about the selected object (Figure 6).

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 6, 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 6, 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:

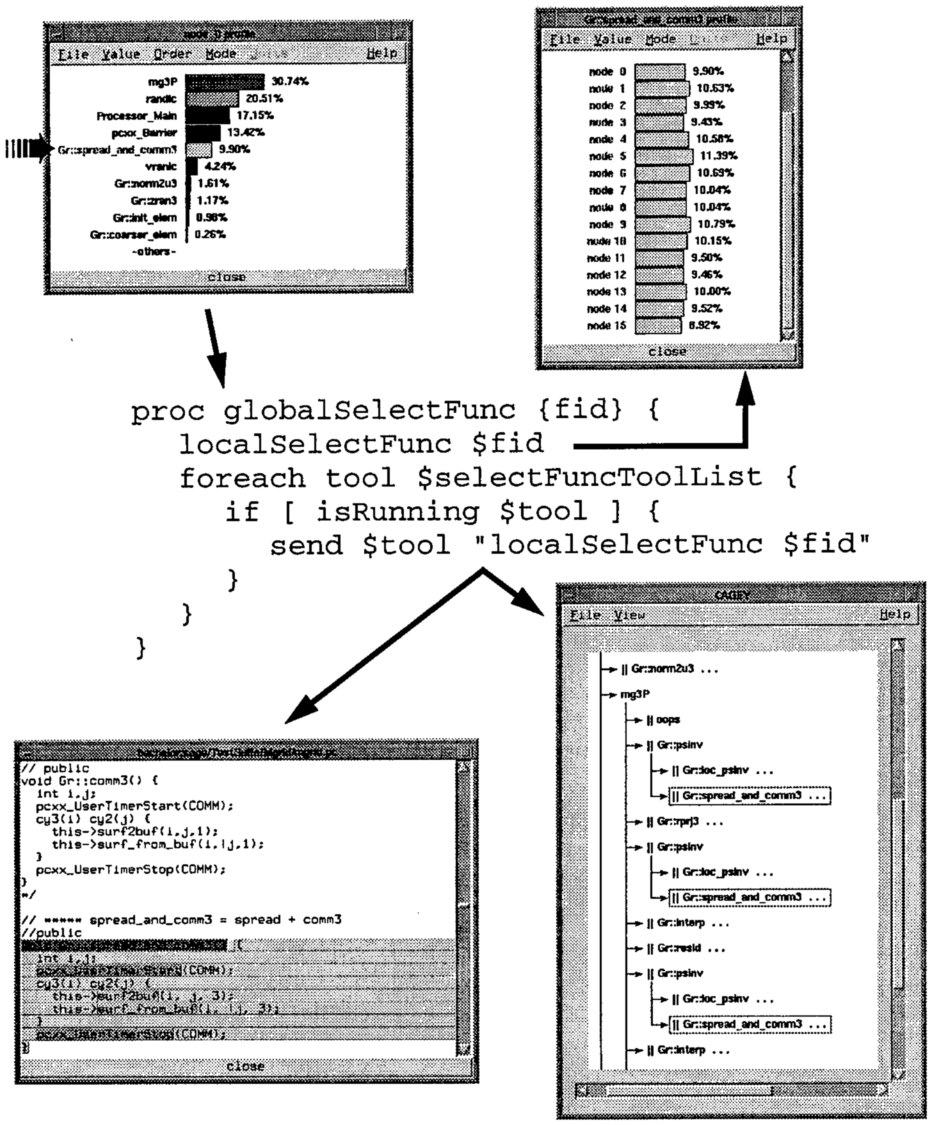
- The use of global features makes it possible to implement the TAU environment as *hypertools*, instead of as a single huge program. Hence, the individual tools can be kept small, which simplifies maintenance and debugging. It also makes tools easier to re-use in different contexts or environments.

- 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.

3.3.4 Internal Tool Interfaces

Figure 7 shows the internal implementation of the TAU's static analysis tools: the source code browser Fancy, the callgraph browser Cagey, and the class hierarchy browser Classy. I DON'T SEE CLASSY IN THIS FIGURE. If a browser is started or switched to another user application, it invokes the object manager



mg3P	30.74%
randic	20.51%
Processor_Main	17.15%
pcxx_Barrier	13.42%
Gr::spread_and_comm3	9.90%
vranic	4.24%
Gr::norm2u3	1.61%
Gr::zran3	1.17%
Gr::init_elem	0.98%
Gr::coarser_elem	0.26%
-others-	

node 0	9.90%
node 1	10.63%
node 2	9.99%
node 3	9.43%
node 4	10.58%
node 5	11.39%
node 6	10.69%
node 7	10.04%
node 8	10.04%
node 9	10.79%
node 10	10.15%
node 11	9.50%
node 12	9.46%
node 13	10.00%
node 14	9.52%
node 15	8.92%

```
proc globalSelectFunc {fid} {
   localSelectFunc $fid
   foreach tool $selectFuncToolList {
      if [ isRunning $tool ] {
         send $tool "localSelectFunc $fid"
      }
   }
}
```

```
// public
void Gr::comm3() {
  int i,j;
  pcxx_UserTimerStart(COMM);
  cy3(i) cy2(j) {
    this->surf2buf(i,j,1);
    this->surf_from_buf(i,|j,1);
  }
  pcxx_UserTimerStop(COMM);
}
*/

// ***** spread_and_comm3 = spread + comm3
//public
[illegible] {
  int i,j;
  pcxx_UserTimerStart(COMM);
  cy3(i) cy2(j) {
    this->surf2buf(i, j, 3);
    this->surf_from_buf(i, |j, 3);
  }
  pcxx_UserTimerStop(COMM);
}
```

```
-> || Gr::norm2u3 ...
-> mg3P
   -> || oops
   -> || Gr::psinv
      -> || Gr::loc_psinv ...
      -> || Gr::spread_and_comm3 ...
   -> || Gr::rprj3 ...
   -> || Gr::psinv
      -> || Gr::loc_psinv ...
      -> || Gr::spread_and_comm3 ...
   -> || Gr::interp ...
   -> || Gr::resid ...
   -> || Gr::psinv
      -> || Gr::loc_psinv ...
      -> || Gr::spread_and_comm3 ...
   -> || Gr::interp ...
```

Figure 6: Global feature Implementation

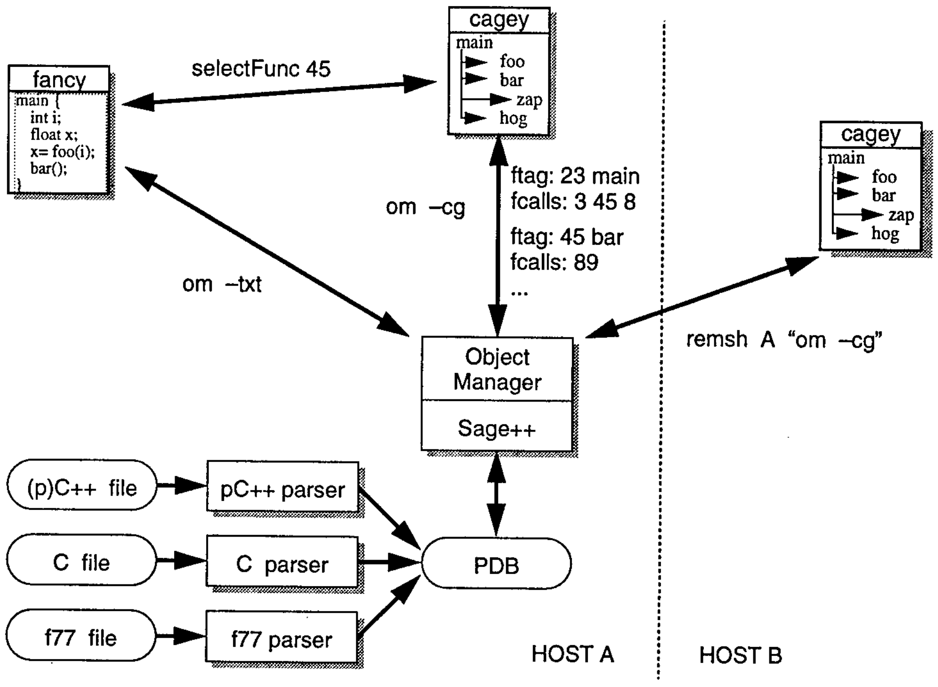


Figure 7: TAU Internal Tool Interfaces

and reads its output. Command line switches 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 in an easy-to-parse ASCII format. The object manager reduces dependence on a particular language system by producing in a generic format where possible, and enhances tool interoperability by making inter-tool communication more robust. For example, this architecture makes it easy to use TAU on a workstation to control a pC++ application running on remote parallel computer. In thise case, instead of launching the object manager directly, the TAU tools use a standard TCP/IP remote shell command.

TAU'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 TAU 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 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 TAU object manager.

4. Change the TAU tool interfaces to the information servers so that they it work with the new language environment.

As an example, TAU 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 TAU-compatible profile data files—to adapt the profile data browser. The whole port required less than one person-week, and shows the benefits of TAU's modular design.

3.4 TAU Tools

In this section, we describe the tools in the current TAU toolset[5]. 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 TAU was designed to support extensions to the toolset, new tools are continually being developed and incorporated into the TAU architecture.

3.4.1 Static Analysis Tools

An earlier version of this material was previously published in [24]. 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.

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

[5]All TAU tools have adjectives as names, so that the answer to "What is TAU?" is "TAU is a Cosy, Fancy, Cagey, Classy, Easy, Racy, Speedy, and Breezy parallel program analysis environment."

3.4.2 File and Source Code Browser

Fancy (File ANd Class displaY) 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 12). 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 13). 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.

3.4.3 Callgraph Browser

Cagey (CAll Graph Extended displaY) shows the static callgraph of the user's application (Figure 14). 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 14 shows a callgraph in compact mode.

3.4.4 Class Hierarchy Browser

Classy (CLASS hierarchY 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 15). 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.

3.4.5 Dynamic Analysis Tools

An earlier version of this material was published in [24].

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. TAU 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.

3.4.6 Program Instrumentation

An earlier version of this section was published in [22].

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 class), 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 to allow to control the instrumentation process for TAU is not yet available.

3.4.7 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. 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 browed 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 16, 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 can be displayed in a variety of formats.

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 21).

3.4.8 Event Tracing for pC++

Some of this material was previously published in [22].

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*; when a program is compiled for tracing, 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.

`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 5: pC++ Event Classes

We have also implemented several utilities for merging event traces, for converting them to other formats, such as Pablo's SDDF [27] or Upshot's ALOG [18], and for analyzing and visualizing traces using the Simple environment or other tools based on the Tdl/Poet interface [23].

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 nesting occurs. WHAT IS "NESTING" IN THIS CONTEXT? Figure 22 shows the major phases of a pC++ program.

3.4.9 Barrier Breakpoint Debugging

An extended version of this section was previously published in [5].

We have developed a program interaction system for pC++ called Breezy

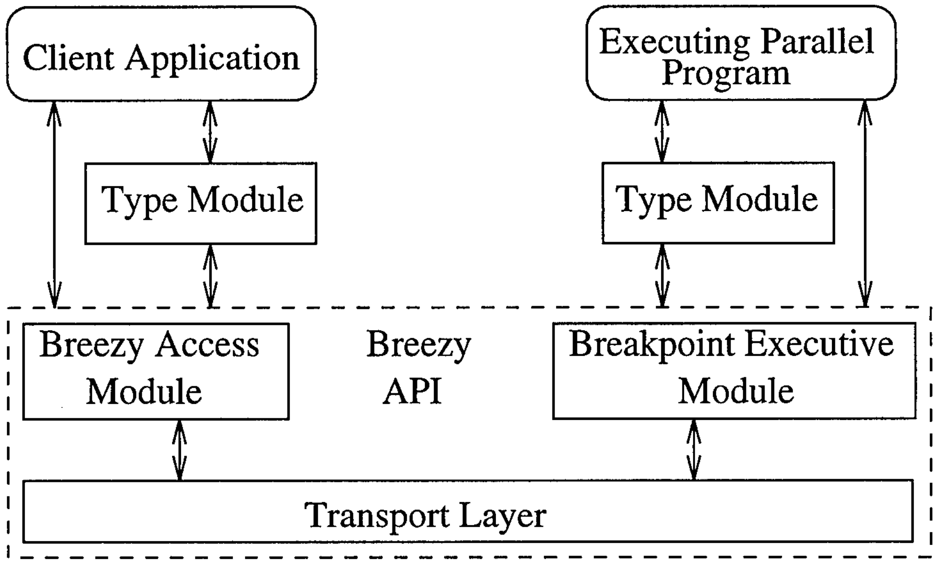


Figure 8: Breezy Architecture

(**BR**eakpoint **E**xecutive **E**nvironment for visuali**Z**ation and data Displa**Y**) [5]. 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 8). The *Breakpoint Executive* maintains information about the program's state, including a list of currently-instantiated parallel data objects. It consults the *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 supports allows a client program 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.

I AM TEMPTED TO CUT THIS ENTIRE LIST, EXCEPT FOR THE LAST POINT. Several unique features of Breezy are: ARE YOU SURE THESE ARE UNIQUE?

- It has an easy-to-use, high-level interface.
- Its modular design allows for component re-use and clean substitution of new technologies (such as replacing the transport layer with CORBA/IDL[25]).
- It can be built on to achieve more complex functionality, such as computational steering. WHAT IS "COMPUTATIONAL STEERING"?
- 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. HOW IS THIS DIFFERENT FROM "COMPUTATIONAL STEERING"?
- Almost all of the implementation is done in the target language. WHAT DOES "ALMOST" MEAN?

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.

3.4.10 Performance Extrapolation for pC++

An extended version of this section was previously published in [2].

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 TAU in the guise of the Speedy tool [2]. 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 an n-processor execution of a pC++ program from its 1-processor execution behavior. Important high-level events, such as remote accesses and barriers, are recorded and timestamped during 1-processor execution. 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 timestampes 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 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 9, and described in more detail in [28, 29, 30].

ExtraP uses pC++'s built-in event tracing system to generate the traces needed for the simulation. These traces can be analyzed using TAU'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 10. 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.

3.5 Tour de TAU: The Polygon Overlay Example

In this section, we show how the TAU environment was used to analyze the pC++ implementation of the polygon overlay problem described in section 2.5.

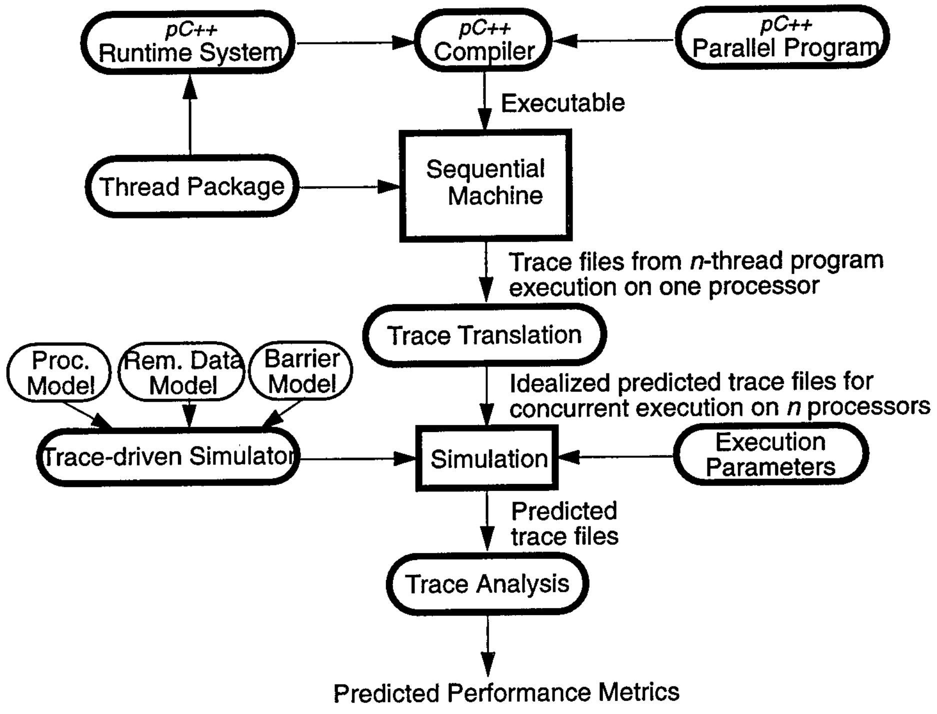
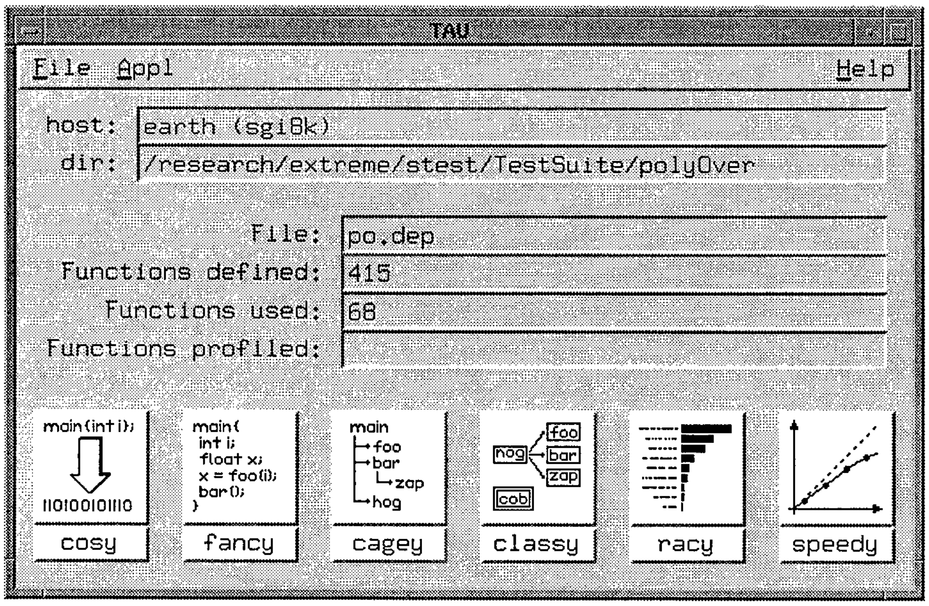


Figure 9: A Performance Extrapolation Technique for pC++



host: earth (sgi8k)
dir: /research/extreme/stest/TestSuite/polyOver
File: po.dep
Functions defined: 415
Functions used: 68
Functions profiled:

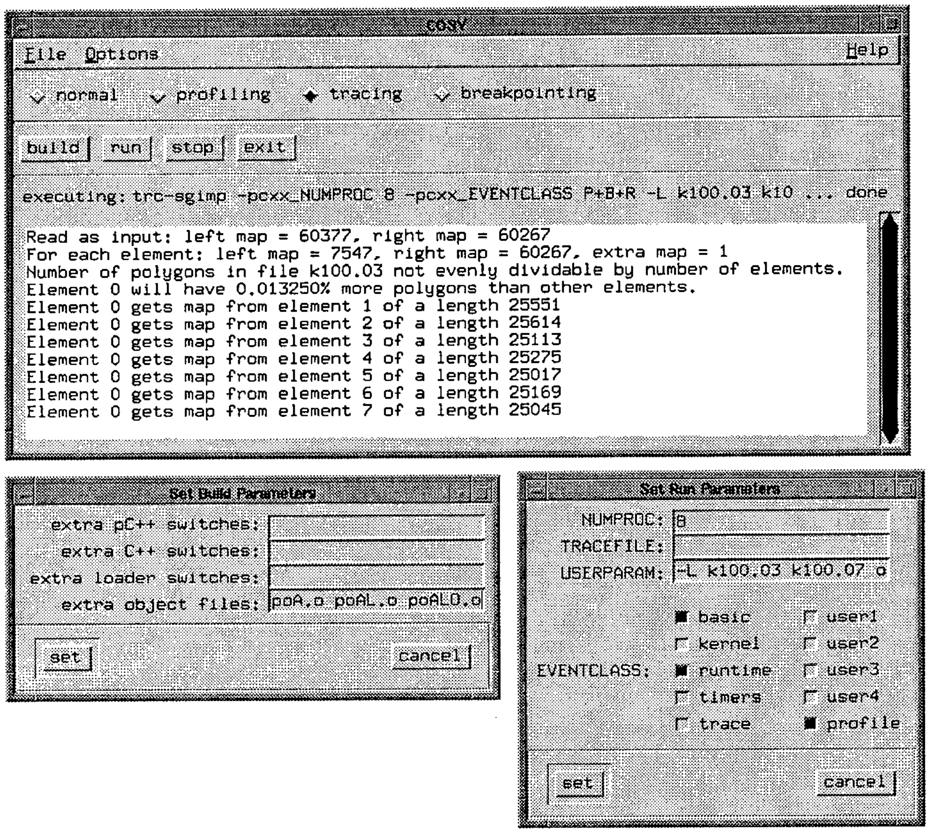
Figure 10: TAU Main Control Panel

3.5.1 Utility Tools

When TAU is started from the command line, the TAU main control panel appears (Figure 10). 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 TAU's static and dynamic tools.

JOINED TWO PARAGRAPHS TO MAKE ONE; PLEASE CHECK. The compilation and execution of pC++ programs can be controlled using Cosy (Figure 11). Cosy automatically connects to the remote machine (if necessary), executes the appropriate commands, and displays the resulting output. Its menu allows users to `build` and `run` pC++ programs, to set compilation and execution parameters, and to do standard tasks like cleaning up or listing the current directory. The `run` button starts the executable that was compiled last and the `stop` button terminates the currently executing command.

Figure 11 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



normal profiling [✓ tracing] breakpointing

```
executing: trc-sgimp -pcxx_NUMPROC 8 -pcxx_EVENTCLASS P+B+R -L k100.03 k10 ... done
```

```
Read as input: left map = 60377, right map = 60267
For each element: left map = 7547, right map = 60267, extra map = 1
Number of polygons in file k100.03 not evenly dividable by number of elements.
Element 0 will have 0.013250% more polygons than other elements.
Element 0 gets map from element 1 of a length 25551
Element 0 gets map from element 2 of a length 25614
Element 0 gets map from element 3 of a length 25113
Element 0 gets map from element 4 of a length 25275
Element 0 gets map from element 5 of a length 25017
Element 0 gets map from element 6 of a length 25169
Element 0 gets map from element 7 of a length 25045
```

extra pC++ switches:
extra C++ switches:
extra loader switches:
extra object files: poA.o poAL.o poALO.o

NUMPROC: 8
TRACEFILE:
USERPARAM: -L k100.03 k100.07 o

EVENTCLASS: [✓ basic] kernel [✓ runtime] timers trace user1 user2 user3 user4 [✓ profile]

Figure 11: Cosy

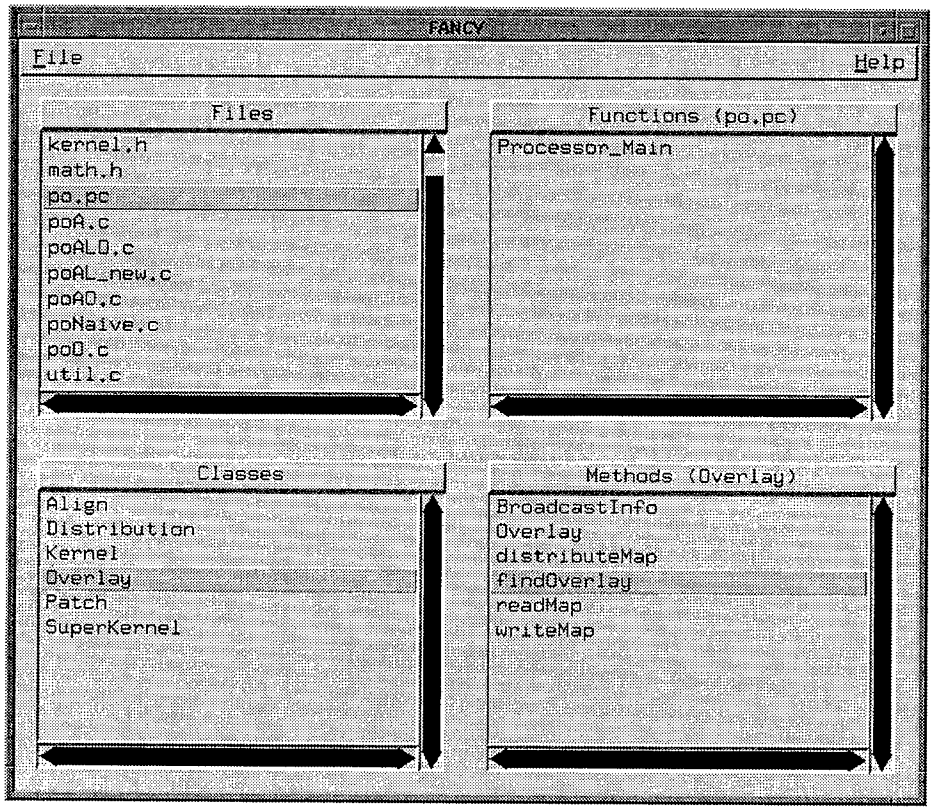
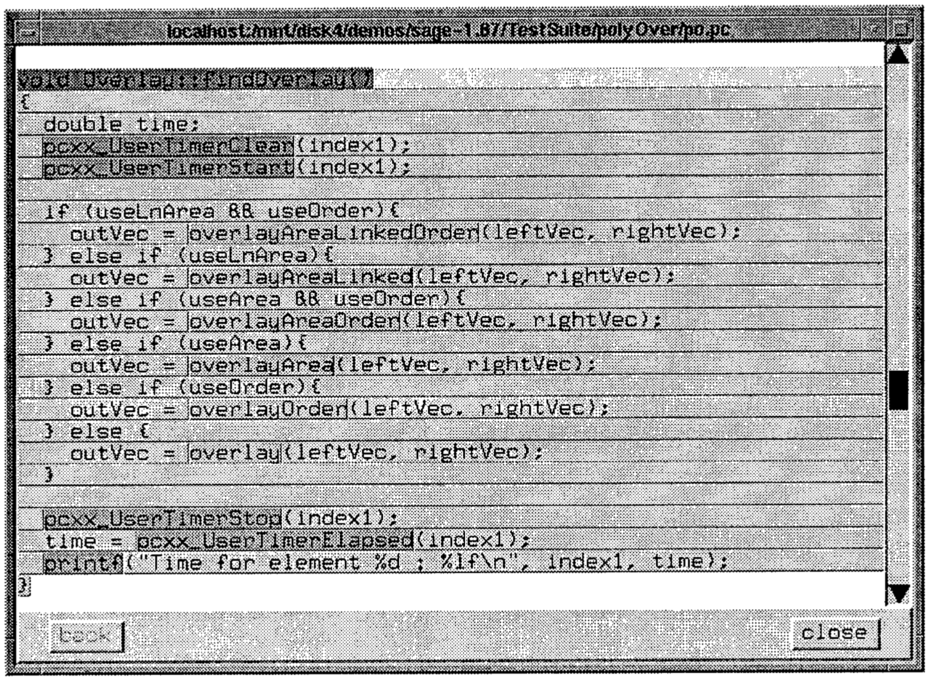


Figure 12: Fancy Main Window

objects in the `Build Parameters` window before we can `build`. The user must supply a makefile for these objects.

3.5.2 Static Analysis Tools

Figure 12 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 (`Overlay`) and element (`Patch`) classes. As described in Section 3.4.2, selecting a file or class shows its components, as shown in Figure 13.


localhost:/mnt/disk4/demos/sage-1.87/TestSuite/polyOver/po.pc
back
close

```
void Overlay::findOverlay()
{
  double time;
  pcxx_UserTimerClear(index1);
  pcxx_UserTimerStart(index1);

  if (useLnArea && useOrder){
    outVec = overlayAreaLinkedOrder(leftVec, rightVec);
  } else if (useLnArea){
    outVec = overlayAreaLinked(leftVec, rightVec);
  } else if (useArea && useOrder){
    outVec = overlayAreaOrder(leftVec, rightVec);
  } else if (useArea){
    outVec = overlayArea(leftVec, rightVec);
  } else if (useOrder){
    outVec = overlayOrder(leftVec, rightVec);
  } else {
    outVec = overlay(leftVec, rightVec);
  }

  pcxx_UserTimerStop(index1);
  time = pcxx_UserTimerElapsed(index1);
  printf("Time for element %d ; %lf\n", index1, time);
}
```

Figure 13: Fancy File Viewer Window

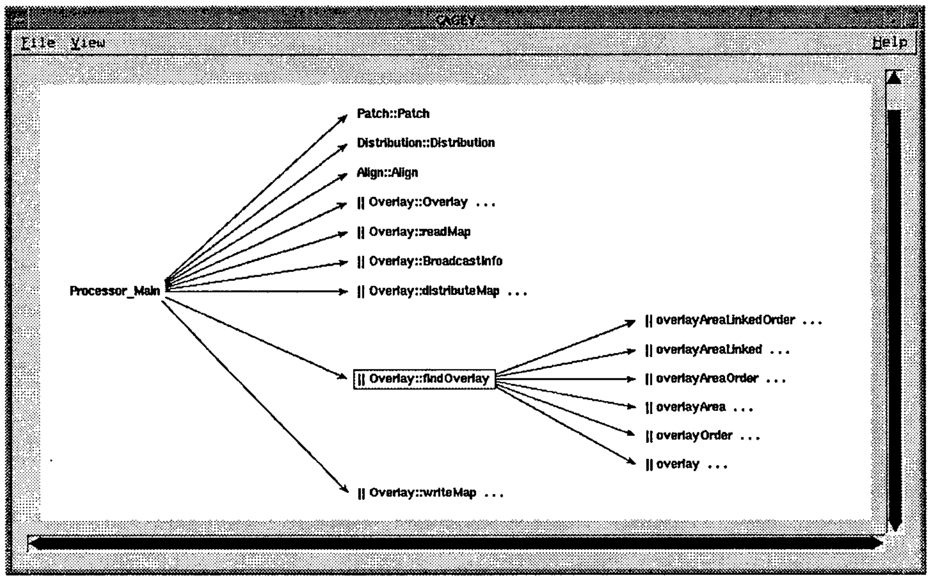


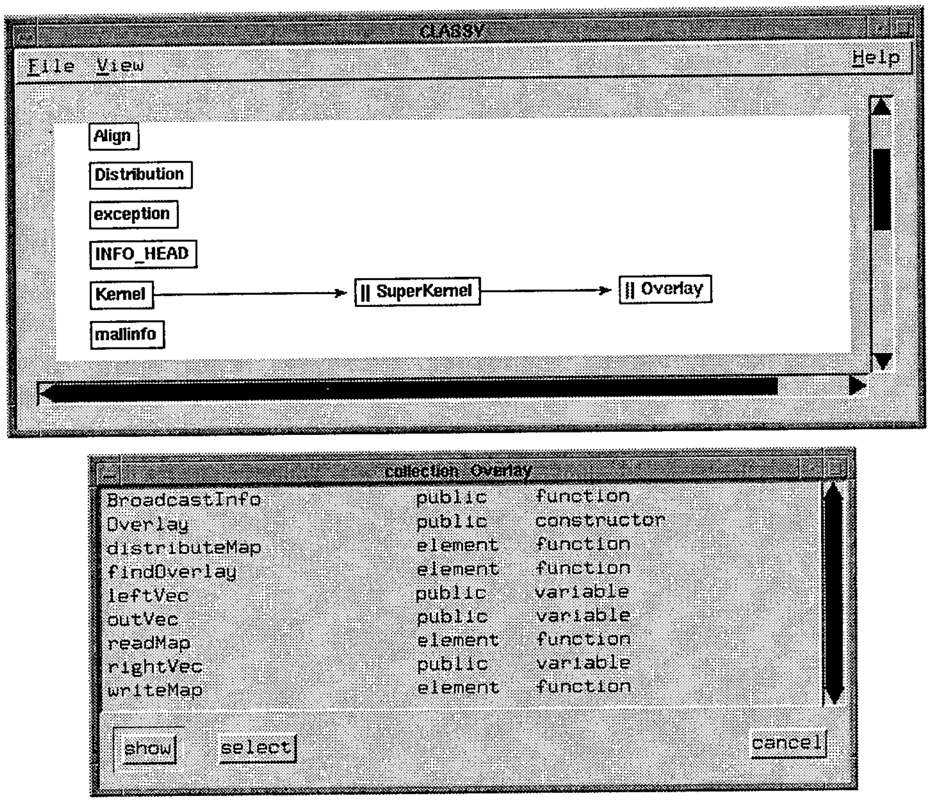
Figure 14: Cagey

Figure 14 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 15). The word `element` is used to indicate the pC++ concept of a *method of element* function. As can be seen, the pC++ polygon overlay algorithm defines its collection class by subclassing from the predefined collection `Superkernel`.

3.5.3 Dynamic Analysis Tools

TAU'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 2.5, 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



```
BroadcastInfo          public    function
Overlay                public    constructor
distributeMap          element   function
findOverlay            element   function
leftVec                public    variable
outVec                 public    variable
readMap                element   function
rightVec               public    variable
writeMap               element   function
```

Figure 15: Classy

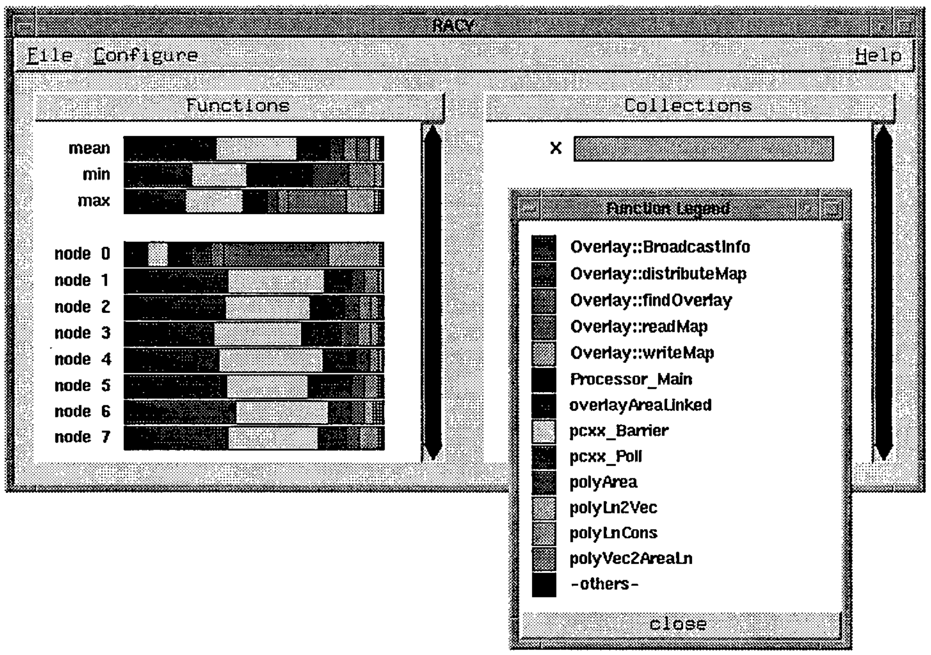


Figure 16: Racy Main Window

input and output phases.

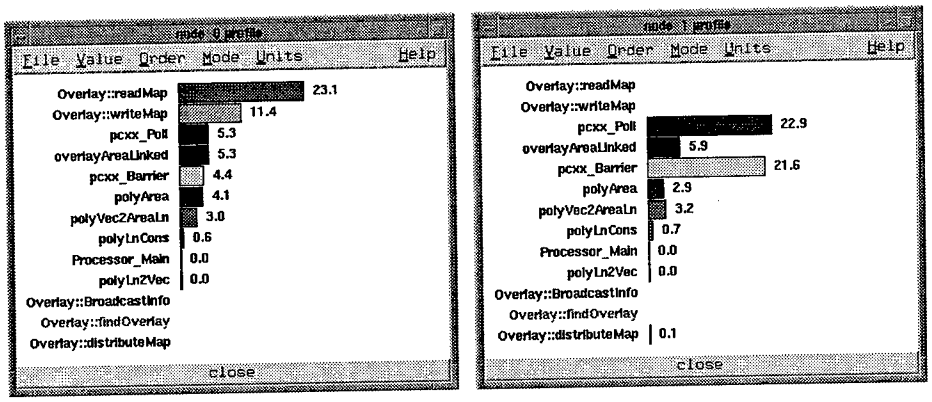
The main window of the Racy profiling tool (Figure 16) gives a quick overview of the application's performance by summarizing both function and collection access profile data. These summary displays allow us to make three important observations about the pC++ polygon overlay program:

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.

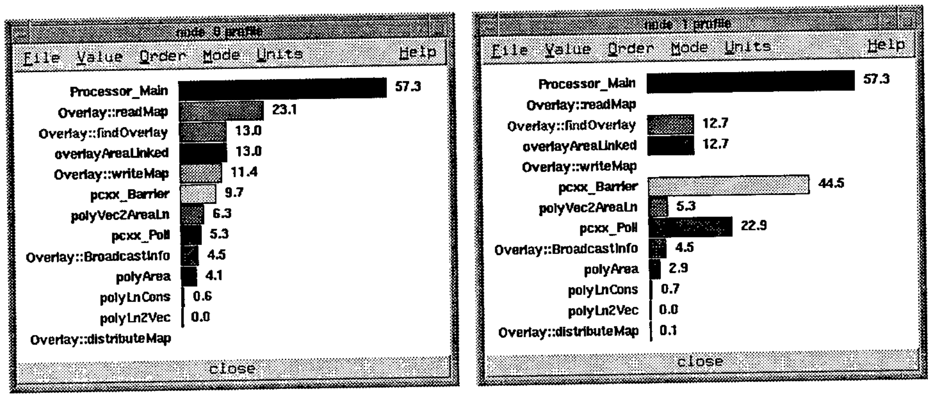
To investigate the program further, we bring up node profiles for node 0 and node 1 (as a representative for the other nodes) by clicking on the labels in the function summary display (Figure 17). As the behavior of node 0 is different



```
Overlay::readMap        23.1
Overlay::writeMap       11.4
pcxx_Poll                5.3
overlayAreaLinked        5.3
pcxx_Barrier             4.4
polyArea                 4.1
polyVec2AreaLn           3.0
polyLnCons               0.6
Processor_Main           0.0
polyLn2Vec               0.0
Overlay::BroadcastInfo
Overlay::findOverlay
Overlay::distributeMap
```

```
Overlay::readMap
Overlay::writeMap
pcxx_Poll               22.9
overlayAreaLinked        5.9
pcxx_Barrier            21.6
polyArea                 2.9
polyVec2AreaLn           3.2
polyLnCons               0.7
Processor_Main           0.0
polyLn2Vec               0.0
Overlay::BroadcastInfo
Overlay::findOverlay
Overlay::distributeMap   0.1
```

Figure 17: Racy Node Profile (showing seconds)



```
Processor_Main          57.3
Overlay::readMap        23.1
Overlay::findOverlay    13.0
overlayAreaLinked       13.0
Overlay::writeMap       11.4
pcxx_Barrier             9.7
polyVec2AreaLn           6.3
pcxx_Poll                5.3
Overlay::BroadcastInfo   4.5
polyArea                 4.1
polyLnCons               0.6
polyLn2Vec               0.0
Overlay::distributeMap
```

```
Processor_Main          57.3
Overlay::readMap
Overlay::findOverlay    12.7
overlayAreaLinked       12.7
Overlay::writeMap
pcxx_Barrier            44.5
polyVec2AreaLn           5.3
pcxx_Poll               22.9
Overlay::BroadcastInfo   4.5
polyArea                 2.9
polyLnCons               0.7
polyLn2Vec               0.0
Overlay::distributeMap   0.1
```

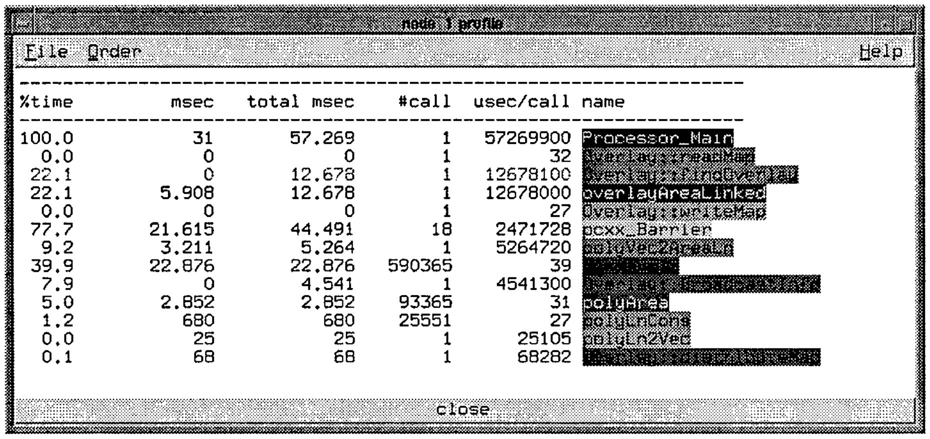
Figure 18: Racy Node Profile (showing total seconds)

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[6] to show the execution time including all children (Figure 18). 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

[6]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.



```
-------------------------------------------------------------------
%time        msec    total msec     #call   usec/call name
-------------------------------------------------------------------
100.0          31        57.269         1    57269900 Processor_Main
  0.0           0             0         1          32 Overlay::readMap
 22.1           0        12.678         1    12678100 Overlay::findOverlay
 22.1       5.908        12.678         1    12678000 overlayAreaLinked
  0.0           0             0         1          27 Overlay::writeMap
 77.7      21.615        44.491        18     2471728 pcxx_Barrier
  9.2       3.211         5.264         1     5264720 polyVec2AreaLn
 39.9      22.876        22.876    590365          39 [illegible]
  7.9           0         4.541         1     4541300 [illegible]
  5.0       2.852         2.852     93365          31 polyArea
  1.2         680           680     25551          27 polyLnCons
  0.0          25            25         1       25105 polyLn2Vec
  0.1          68            68         1       68282 [illegible]
```

Figure 19: Racy Text Node Profile

representing the function. Figure 20 shows the time spent in `Overlay::findOverlay` and its children on the different nodes used for the execution of the program.

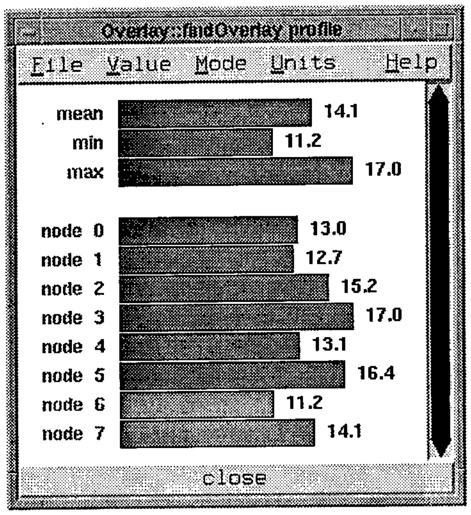
This information lets us hypothesize that most of the waiting on processors 1 to 7 happens during the I/O phase on processor 0. Simple profiling does not allow us to verify our hypothesis, but we can do this very easily using event tracing.

After the execution of the program, 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 22). 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 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.

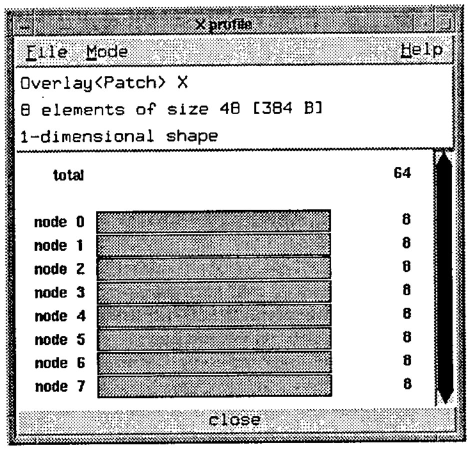
The TAU barrier breakpoint debugger, Breezy, allows users to control a pC++ program running on a parallel machine from a remote workstation. Unfortunately, the polygon overlay example is too simple to allow a full demonstration of Breezy's functionality.

Having used Cosy to compile the program for breakpointing, executing it automatically launches the Breezy main control panel (Figure 23). 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



mean	14.1
min	11.2
max	17.0
node 0	13.0
node 1	12.7
node 2	15.2
node 3	17.0
node 4	13.1
node 5	16.4
node 6	11.2
node 7	14.1

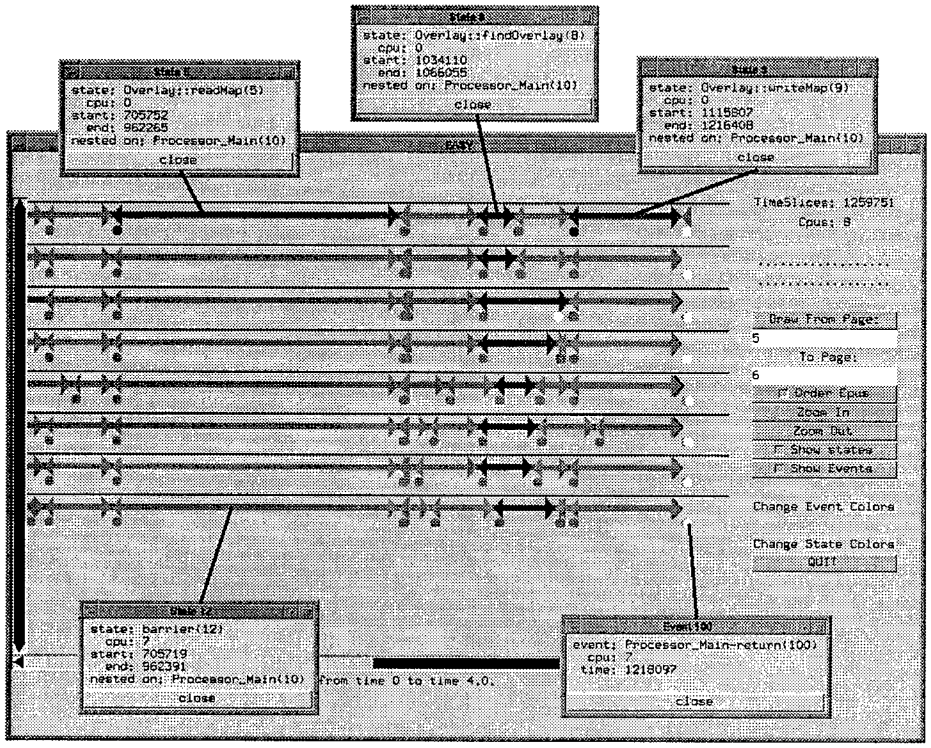
Figure 20: Racy Function Profile



```
Overlay<Patch> X
8 elements of size 48 [384 B]
1-dimensional shape
```

total	64
node 0	8
node 1	8
node 2	8
node 3	8
node 4	8
node 5	8
node 6	8
node 7	8

Figure 21: Racy Collection Profile



```
state: Overlay::readMap(5)
  cpu: 0
start: 705752
  end: 962265
nested on: Processor_Main(10)
```

```
state: Overlay::findOverlay(8)
  cpu: 0
start: 1034110
  end: 1066055
nested on: Processor_Main(10)
```

```
state: Overlay::writeMap(9)
  cpu: 0
start: 1115807
  end: 1216408
nested on: Processor_Main(10)
```

```
TimeSlices: 1259751
    Cpus: 8
```

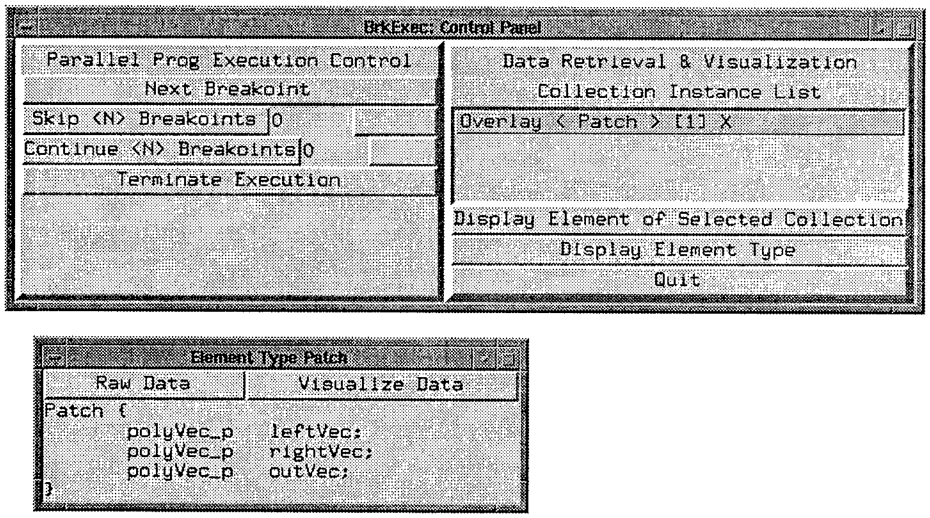
Draw From Page: 5
To Page: 6

```
state: barrier(12)
  cpu: 7
start: 705719
  end: 962391
nested on: Processor_Main(10)
```

from time 0 to time 4.0.

```
event: Processor_Main-return(100)
  cpu: 7
 time: 1218097
```

Figure 22: Easy



Skip <N> Breakoints 0
Continue <N> Breakoints 0

Overlay < Patch > [1] X

```
Patch {
        polyVec_p   leftVec;
        polyVec_p   rightVec;
        polyVec_p   outVec;
}
```

Figure 23: Breezy

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.

MAJOR SURGERY ON THIS PARAGRAPH. 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 TAU release.

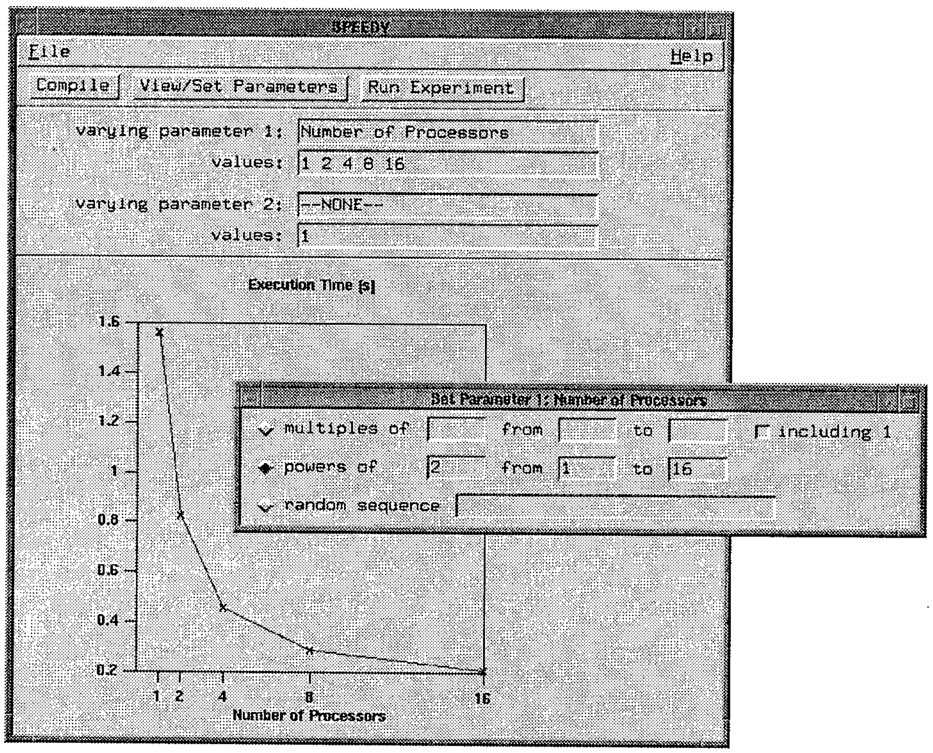
3.5.4 Performance Extrapolation

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 3.4.10. 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 24) is launched by clicking the Speedy button on the TAU 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 25) 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 one of the 25 ExtraP parameters which is significant is `MipsRatio`, simulation (HP) and actual (SGI) machine 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`.

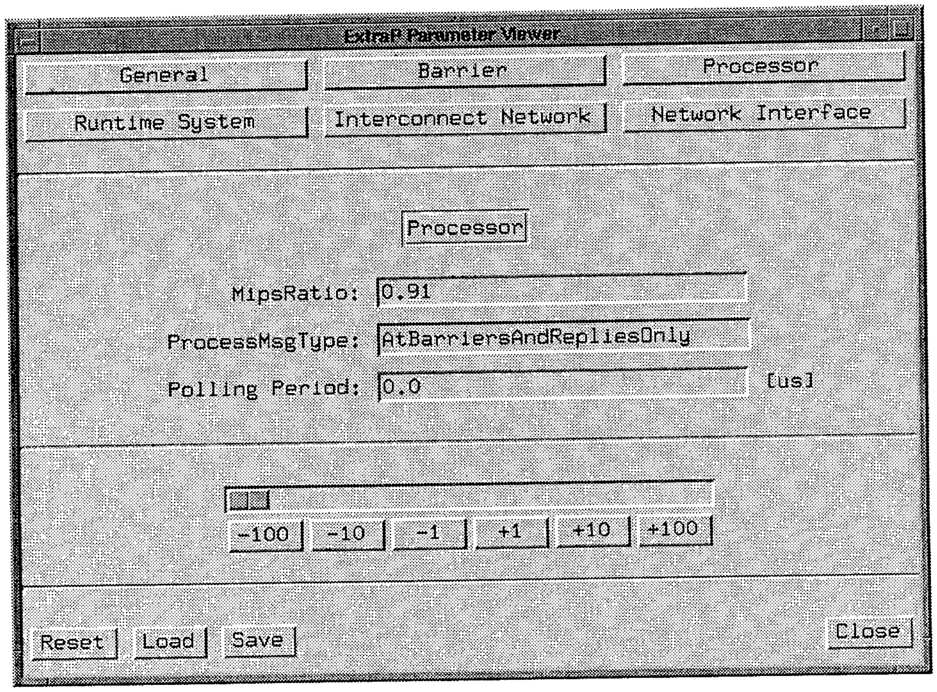
Clicking Speedy's `Run Experiment` button starts the experiment. After each iteration of the extrapolation, the execution time graph in the Speedy main window is updated, which allows the user to see, and control, the evolution of the experiment. This is important for long-running experiments: if something 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 26b). If the varying parameter is `Number of Processors`, as in our example, a speedup



varying parameter 1: Number of Processors
values: 1 2 4 8 16
varying parameter 2: --NONE--
values: 1

multiples of from to including 1
[✓ powers of 2 from 1 to 16]
random sequence

Figure 24: Speedy Main Window



MipsRatio: 0.91
ProcessMsgType: AtBarriersAndRepliesOnly
Polling Period: 0.0 [us]

Figure 25: Speedy Parameter Viewer

display is also available (Figure 26a).

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 6: Execution Time in Seconds

As Speedy is fully integrated with TAU, users can employ other TAU tools to verify extrapolation results, or compare them with actual measurements. Table 6 lists both measured and estimated execution times for our SGI PowerChallenge. As can be seen, the two sets of values are reasonably close.

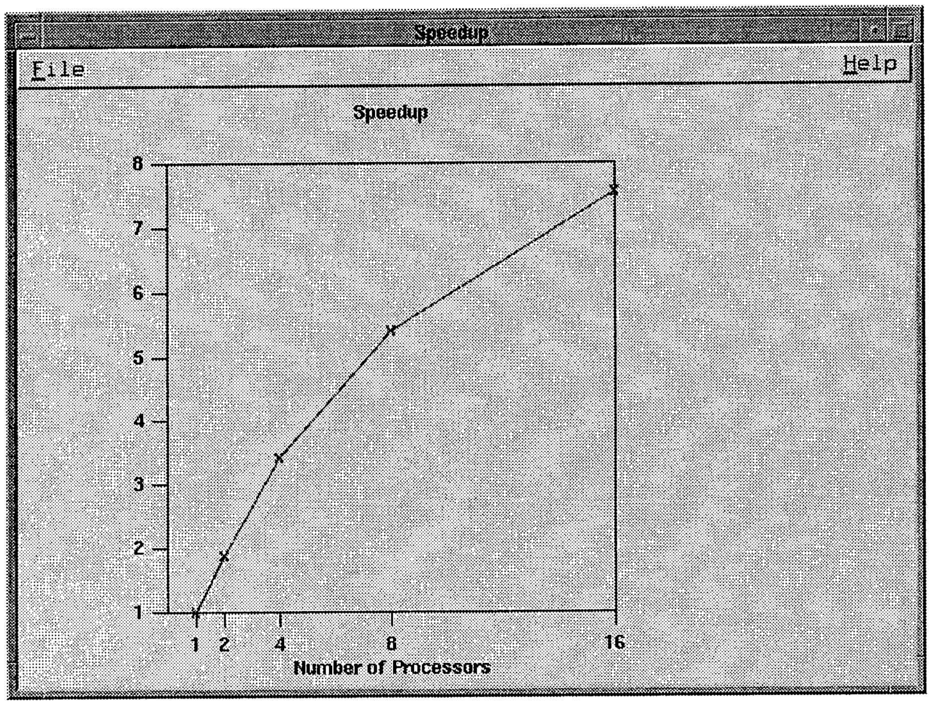
3.6 Critique

In this section, we evaluate TAU based on its implementation on a large number of parallel platforms.

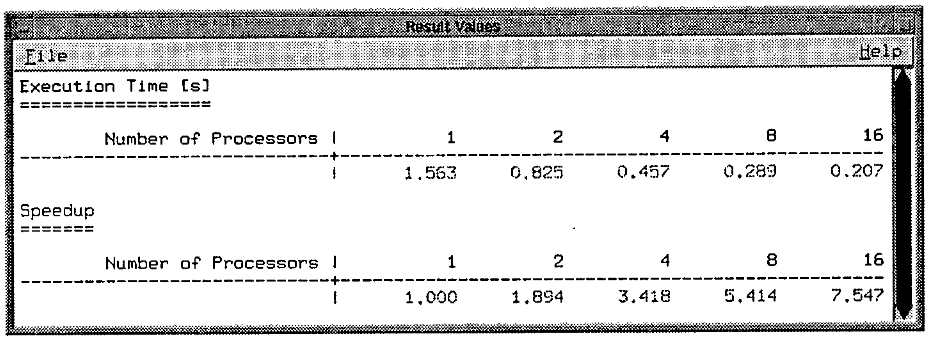
- *A parallel programming environment should support the full development circle.* The TAU environment currently supports compilation and execution control, static and dynamic program and performance analysis, debugging, and performance extrapolation. One area which is not yet supported is program development. Currently, we are working on an editor which would be integrated with the other TAU tools. This would replace the Fancy source code viewer. 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, TAU currently only supports function and class symbol lookup. We are planning to enhance the TAU 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) 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 26: Speedy Result Values Windows

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 TAU, which, like pC++, runs on every major commercial parallel computer and UNIX workstation and work 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.

3.7 Conclusion and Future Work

The current TAU system has been highly successful in meeting the program analysis requirements of the pC++ language system. However, TAU 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. TAU'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, TAU 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 TAU tools will be modified to work transparently during replay.

Replay, however, is only one approach, and is quite expensive in tracing overhead. TAU 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 [20]. 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 [20].

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, TAU 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.

References

[1] V.S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D.A. Reed. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing 1995*, December 1995.

[2] H. Beilner and F. Bause, editors. *Speedy: An Integrated Performance Extrapolation Tool for pC++ Programs*, number 977 in Lecture Notes in Computer Science. Springer–Verlag, September 1995.

[3] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An Object Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In M. Chapman and A. Vermeulen, editors, *Proceedings of the Second Annual Object-Oriented Numerics Conference*, Corvallis, OR, 1994. Rogue Wave Software.

[4] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), 1993.

[5] D. Brown, A. Malony, and B. 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.

[6] M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, and Y. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data*, May 1994.

[7] K.M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-oriented Programming Notation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 281–313. MIT Press, 1993. ISBN 0-272-01139-5.

[8] J. Choi, J.J. Dongarra, and D.W. Walker. The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers. In H.J. Siegel, editor, *Proc. Eighth International Parallel Processing Symposium*. IEEE Computer Society Press, April 1994.

[9] R. Ferrell and E. Bertschinger. Particle-Mesh Methods on the Connection Machine. *International Journal of Modern Physics C*, 1993.

[10] D. Gannon. Libraries and Tools for Object Parallel Programming. In *Advances in Parallel Computing: CNRS-NSF Workshop on Environments and*

Tools for Parallel Scientific Computing, Saint Hilaire du Touvet, volume 6, pages 231–246. Elsevier Science Publisher, 1993.

[11] D. Gannon, V. Guarna, and J.-K. Lee. Static Analysis and Runtime Support for Parallel Execution of C. In *Languages and Compilers for Parallel Computing*, pages 254–274. MIT Press, 1993.

[12] D. Gannon and J.-K. Lee. Object Oriented Parallelism: pC++ Ideas and Experiments. In *Japan Society for Parallel Processing*, pages 13–23, 1991.

[13] D. Gannon and J.-K. Lee. On Using Object Oriented Parallel Programming to Build Distributed Algebraic Abstractions. In Bourge and Cosnard, editors, *Proceedings of CONPAR 92–VAPP V*, pages 769–774. Springer Verlag, 1992.

[14] D. Gannon, N. Sundaresan, and P. Beckman. pC++ Meets Multithreaded Computation. In J.J. Dongarra and B. Tourancheau, editors, *Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 76–85. SIAM Press, 1994.

[15] D. Gannon, S. Yang, P. Bode, and V. Menkov. Object Oriented Methods for Parallel Execution of Astrophysics Simulations. In *Mardigras Teraflops Grand Challenge Conference*. Lousiana State University, 1994.

[16] J. Gotwals, S. Srinivas, and D. Gannon. pC++/streams: a Library for I/O on Complex Distributed Data Structures. In *Symposium on the Principles and Practice of Parallel Programming*. ACM, 1995.

[17] L. Hernquist and J.P. Ostriker. A Self-Consistent Field Method for Galactic Dynamics. *The Astrophysical Journal*, 386:375–397, 1992.

[18] 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.

[19] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

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

[21] M. Lemke and D. Quinlan. P++, a Parallel C++ Array Class Library for Architecture-Independent Development of Numerical Software. In *Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 268–269, 1993.

[22] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers. In H.J. Siegel, editor, *Proc. Eighth International Parallel Processing Symposium.* IEEE Computer Society Press, April 1994.

[23] B. 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 103–124. Elsevier, September 1992.

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

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

[26] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[27] 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 562–569. IEEE Computer Society Press, 1991.

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

[29] K. Shanmugam and A. Malony. Performance Extrapolation of Parallel Programs. In C. Polychronopoulos, editor, *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, volume II Software, pages 117–120. CRC Press, August 1995.

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

[31] B. Stroustrup. Parameterized Types for C++. In *USENIX C++ Conference, Denver*, October 1988.

[32] S. Yang, D. Gannon, S. Bodin, P. Bode, and S. Srinivas. High Performance Fortran Interface to the Parallel C++. In *Scalable High Performance Computing Conference.* IEEE, 1994.

☆U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61241

DISTRIBUTION LIST

addresses	number of copies
DR. RAYMOND A. LIUZZI AFRL/IFTB 525 BROOKS ROAD ROME, NY 13441-4505	10
DENNIS GANNON INDIANA UNIVERSITY SPONSORED RESEARCH SERVICE P.O. BOX 1847 BLOOMINGTON, IN 47405-4101	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
RELIABILITY ANALYSIS CENTER 201 MILL ST. ROME NY 13440-8200	1
ATTN: GWEN NGUYEN GIDEP P.O. BOX 8000 CORONA CA 91718-8000	1
AFIT ACADEMIC LIBRARY/LDEE 2950 P STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1

ATTN: GILBERT G. KUPERMAN AL/CFHI, BLDG. 248 2255 H STREET WRIGHT-PATTERSON AFB OH 45433-7022	1
ATTN: TECHNICAL DOCUMENTS CENTER OL AL HSC/HRG 2698 G STREET WRIGHT-PATTERSON AFB OH 45433-7604	1
AIR UNIVERSITY LIBRARY (AUL/LSAD) 600 CHENNAULT CIRCLE MAXWELL AFB AL 36112-6424	1
US ARMY SSDC P.O. BOX 1500 ATTN: CSSD-IM-PA HUNTSVILLE AL 35807-3801	1
TECHNICAL LIBRARY D0274(PL-TS) SPAWARSYSCEN 53560 HULL STREET SAN DIEGO CA 92152-5001	1
NAVAL AIR WARFARE CENTER WEAPONS DIVISION CODE 4BL000D 1 ADMINISTRATION CIRCLE CHINA LAKE CA 93555-6100	1
SPACE & NAVAL WARFARE SYSTEMS CMD ATTN: PMW163-1 (R. SKIANO)RM 1044A 53560 HULL ST. SAN DIEGO, CA 92152-5002	2
SPACE & NAVAL WARFARE SYSTEMS COMMAND, EXECUTIVE DIRECTOR (PD13A) ATTN: MR. CARL ANDRIANI 2451 CRYSTAL DRIVE ARLINGTON VA 22245-5200	1
COMMANDER, SPACE & NAVAL WARFARE SYSTEMS COMMAND (CODE 32) 2451 CRYSTAL DRIVE ARLINGTON VA 22245-5200	1

CDR, US ARMY MISSILE COMMAND REDSTONE SCIENTIFIC INFORMATION CTR ATTN: AMSMI-RD-CS-R, DOCS REDSTONE ARSENAL AL 35898-5241	2
ADVISORY GROUP ON ELECTRON DEVICES SUITE 500 1745 JEFFERSON DAVIS HIGHWAY ARLINGTON VA 22202	1
REPORT COLLECTION, CIC-14 MS P364 LOS ALAMOS NATIONAL LABORATORY LOS ALAMOS NM 87545	1
AEDC LIBRARY TECHNICAL REPORTS FILE 100 KINDEL DRIVE, SUITE C211 ARNOLD AFB TN 37389-3211	1
COMMANDER USAISC ASHC-IMD-L, BLDG 61801 FT HUACHUCA AZ 85613-5000	1
US DEPT OF TRANSPORTATION LIBRARY FB10A, M-457, RM 930 800 INDEPENDENCE AVE, SW WASH DC 22591	1
AWS TECHNICAL LIBRARY 859 BUCHANAN STREET, RM. 427 SCOTT AFB IL 62225-5118	1
AFIWC/MSY 102 HALL BLVD, STE 315 SAN ANTONIO TX 78243-7016	1
SOFTWARE ENGINEERING INSTITUTE CARNEGIE MELLON UNIVERSITY 4500 FIFTH AVENUE PITTSBURGH PA 15213	1

NSA/CSS 1
K1
FT MEADE MD 20755-6000

ATTN: OM CHAUHAN 1
DCMC WICHITA
271 WEST THIRD STREET NORTH
SUITE 6000
WICHITA KS 67202-1212

AFRL/VSOS-TL (LIBRARY) 1
5 WRIGHT STREET
HANSCOM AFB MA 01731-3004

ATTN: EILEEN LADUKE/D460 1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

OUSD(P)/DTSA/DUTD 2
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

SOFTWARE ENGR'G INST TECH LIBRARY 1
ATTN: MR DENNIS SMITH
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PA 15213-3890

USC-ISI 1
ATTN: DR ROBERT M. BALZER
4676 ADMIRALTY WAY
MARINA DEL REY CA 90292-6695

KESTREL INSTITUTE 1
ATTN: DR CORDELL GREEN
1801 PAGE MILL ROAD
PALO ALTO CA 94304

ROCHESTER INSTITUTE OF TECHNOLOGY 1
ATTN: PROF J. A. LASKY
1 LOMB MEMORIAL DRIVE
P.O. BOX 9887
ROCHESTER NY 14613-5700

AFIT/ENG 1
ATTN:TOM HARTRUM
WPAFB OH 45433-6583

THE MITRE CORPORATION 1
ATTN: MR EDWARD H. BENSLEY
BURLINGTON RD/MAIL STOP A350
BEDFORD MA 01730

UNIV OF ILLINOIS, URBANA-CHAMPAIGN 1
ATTN: ANDREW CHIEN
DEPT OF COMPUTER SCIENCES
1304 W. SPRINGFIELD/240 DIGITAL LAB
URBANA IL 61801

HONEYWELL, INC. 1
ATTN: MR BERT HARRIS
FEDERAL SYSTEMS
7900 WESTPARK DRIVE
MCLEAN VA 22102

SOFTWARE ENGINEERING INSTITUTE 1
ATTN: MR WILLIAM E. HEFLEY
CARNEGIE-MELLON UNIVERSITY
SEI 2218
PITTSBURGH PA 15213-38990

UNIVERSITY OF SOUTHERN CALIFORNIA 1
ATTN: DR. YIGAL ARENS
INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY/SUITE 1001
MARINA DEL REY CA 90292-6695

COLUMBIA UNIV/DEPT COMPUTER SCIENCE 1
ATTN: DR GAIL E. KAISER
450 COMPUTER SCIENCE BLDG
500 WEST 120TH STREET
NEW YORK NY 10027

SOFTWARE PRODUCTIVITY CONSORTIUM 1
ATTN: MR ROBERT LAI
2214 ROCK HILL ROAD
HERNDON VA 22070

AFIT/ENG 1
ATTN: DR GARY B. LAMONT
SCHOOL OF ENGINEERING
DEPT ELECTRICAL & COMPUTER ENGRG
WPAFB OH 45433-6583

NSA/OFC OF RESEARCH 1
ATTN: MS MARY ANNE OVERMAN
9800 SAVAGE ROAD
FT GEORGE G. MEADE MD 20755-6000

AT&T BELL LABORATORIES 1
ATTN: MR PETER G. SELFRIDGE
ROOM 3C-441
600 MOUNTAIN AVE
MURRAY HILL NJ 07974

ODYSSEY RESEARCH ASSOCIATES, INC. 1
ATTN: MS MAUREEN STILLMAN
301A HARRIS B. DATES DRIVE
ITHACA NY 14850-1313

TEXAS INSTRUMENTS INCORPORATED 1
ATTN: DR DAVID L. WELLS
P.O. BOX 655474, MS 238
DALLAS TX 75265

TEXAS A & M UNIVERSITY 1
ATTN: DR PAULA MAYER
KNOWLEDGE BASED SYSTEMS LABORATORY
DEPT OF INDUSTRIAL ENGINEERING
COLLEGE STATION TX 77843

KESTREL DEVELOPMENT CORPORATION 1
ATTN: DR RICHARD JULLIG
3260 HILLVIEW AVENUE
PALO ALTO CA 94304

DARPA/ITO 1
ATTN: DR KIRSTIE BELLMAN
3701 N FAIRFAX DRIVE
ARLINGTON VA 22203-1714

NASA/JOHNSON SPACE CENTER 1
ATTN: CHRIS CULBERT
MAIL CODE PT4
HOUSTON TX 77058

SAIC 1
ATTN: LANCE MILLER
MS T1-6-3
PO BOX 1303 (OR 1710 GOODRIDGE DR)
MCLEAN VA 22102

STERLING IMD INC. 1
KSC OPERATIONS
ATTN: MARK MAGINN
BEECHES TECHNICAL CAMPUS/RT 26 N.
ROME NY 13440

NAVAL POSTGRADUATE SCHOOL 1
ATTN: BALA RAMESH
CODE AS/RS
ADMINISTRATIVE SCIENCES DEPT
MONTEREY CA 93943

HUGHES SPACE & COMMUNICATIONS 1
ATTN: GERRY BARKSDALE
P. O. BOX 92919
BLDG R11 MS M352
LOS ANGELES, CA 90009-2919

SCHLUMBERGER LABORATORY FOR 1
COMPUTER SCIENCE
ATTN: DR. GUILLERMO ARANGO
8311 NORTH FM620
AUSTIN, TX 78720

DECISION SYSTEMS DEPARTMENT 1
ATTN: PROF WALT SCACCHI
SCHOOL OF BUSINESS
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CA 90089-1421

SOUTHWEST RESEARCH INSTITUTE 1
ATTN: BRUCE REYNOLDS
6220 CULEBRA ROAD
SAN ANTONIO, TX 78228-0510

NATIONAL INSTITUTE OF STANDARDS 1
AND TECHNOLOGY
ATTN: CHRIS DABROWSKI
ROOM A266, BLDG 225
GAITHSBURG MD 20899

EXPERT SYSTEMS LABORATORY 1
ATTN: STEVEN H. SCHWARTZ
NYNEX SCIENCE & TECHNOLOGY
500 WESTCHESTER AVENUE
WHITE PLAINS NY 20604

NAVAL TRAINING SYSTEMS CENTER 1
ATTN: ROBERT BREAUX/CODE 252
12350 RESEARCH PARKWAY
ORLANDO FL 32826-3224

CENTER FOR EXCELLENCE IN COMPUTER- AIDED SYSTEMS ENGINEERING ATTN: PERRY ALEXANDER 2291 IRVING HILL ROAD LAWRENCE KS 66049	1
DR JOHN SALASIN DARPA/ITO 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
DR BARRY BOEHM DIR, USC CENTER FOR SW ENGINEERING COMPUTER SCIENCE DEPT UNIV OF SOUTHERN CALIFORNIA LOS ANGELES CA 90089-0781	1
DR STEVE CROSS CARNEGIE MELLON UNIVERSITY SCHOOL OF COMPUTER SCIENCE PITTSBURGH PA 15213-3891	1
DR MARK MAYBURY MITRE CORPORATION ADVANCED INFO SYS TECH; G041 BURLINTON ROAD, M/S K-329 BEDFORD MA 01730	1
ISX ATTN: MR. SCOTT FOUSE 4353 PARK TERRACE DRIVE WESTLAKE VILLAGE,CA 91361	1
MR GARY EDWARDS ISX 433 PARK TERRACE DRIVE WESTLAKE VILLAGE CA 91361	1
DR ED WALKER BBN SYSTEMS & TECH CORPORATION 10 MOULTON STREET CAMBRIDGE MA 02238	1
LEE ERMAN CIMFLEX TEKNOWLEDGE 1810 EMBACADERO ROAD P.O. BOX 10119 PALO ALTO CA 94303	1

DR. DAVE GUNNING 1
DARPA/ISO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

DAN WELD 1
UNIVERSITY OF WASHINGTON
DEPART OF COMPUTER SCIENCE & ENGIN
BOX 352350
SEATTLE, WA 98195-2350

STEPHEN SODERLAND 1
UNIVERSITY OF WASHINGTON
DEPT OF COMPUTER SCIENCE & ENGIN
BOX 352350
SEATTLE, WA 98195-2350

DR. MICHAEL PITTARELLI 1
COMPUTER SCIENCE DEPART
SUNY INST OF TECH AT UTICA/ROME
P.O. BOX 3050
UTICA, NY 13504-3050

CAPRARO TECHNOLOGIES, INC 1
ATTN: GERARD CAPRARO
311 TURNER ST.
UTICA, NY 13501

USC/ISI 1
ATTN: BOB MCGREGOR
4676 ADMIRALTY WAY
MARINA DEL REY, CA 90292

SRI INTERNATIONAL 1
ATTN: ENRIQUE RUSPINI
333 RAVENSWOOD AVE
MENLO PARK, CA 94025

DARTMOUTH COLLEGE 1
ATTN: DANIELA RUS
DEPT OF COMPUTER SCIENCE
11 ROPE FERRY ROAD
HANOVER, NH 03755-3510

UNIVERSITY OF FLORIDA 1
ATTN: ERIC HANSON
CISE DEPT 456 CSE
GAINESVILLE, FL 32611-6120

CARNEGIE MELLON UNIVERSITY ATTN: TOM MITCHELL COMPUTER SCIENCE DEPARTMENT PITTSBURGH, PA 15213-3890	1
CARNEGIE MELLON UNIVERSITY ATTN: MARK CRAVEN COMPUTER SCIENCE DEPARTMENT PITTSBURGH, PA 15213-3890	1
UNIVERSITY OF ROCHESTER ATTN: JAMES ALLEN DEPARTMENT OF COMPUTER SCIENCE ROCHESTER, NY 14627	1
TEXTWISE, LLC ATTN: LIZ LIDDY 2-121 CENTER FOR SCIENCE & TECH SYRACUSE, NY 13244	1
WRIGHT STATE UNIVERSITY ATTN: DR. BRUCE BERRA DEPART OF COMPUTER SCIENCE & ENGIN DAYTON, OHIO 45435-0001	1
UNIVERSITY OF FLORIDA ATTN: SHARMA CHAKRAVARTHY COMPUTER & INFOR SCIENCE DEPART GAINESVILLE, FL 32622-6125	1
KESTREL INSTITUTE ATTN: DAVID ESPINOSA 3260 HILLVIEW AVENUE PALO ALTO, CA 94304	1
STOLLER-HENKE ASSOCIATES ATTN: T.J. GOAN 2016 BELLE MONTI AVENUE BELMONT, CA 94002	1
USC/INFORMATION SCIENCE INSTITUTE ATTN: DR. CARL KESSELMAN 11474 ADMIRALTY WAY, SUITE 1001 MARINA DEL REY, CA 90292	1

MASSACHUSETTS INSTITUTE OF TECH ATTN: DR. MICHAELE SIEGEL SLOAN SCHOOL 77 MASSACHUSETTS AVENUE CAMBRIDGE, MA 02139	1
USC/INFORMATION SCIENCE INSTITUTE ATTN: DR. WILLIAM SWARTHOUT 11474 ADMIRALTY WAY, SUITE 1001 MARINA DEL REY, CA 90292	1
STANFORD UNIVERSITY ATTN: DR. GIO WIEDERHOLD 857 SIERRA STREET STANFORD SANTA CLARA COUNTY, CA 94305-4125	1
NCCOSC RDTE DIV D44208 ATTN: LEAH WONG 53245 PATTERSON ROAD SAN DIEGO, CA 92152-7151	1
SPAWAR SYSTEM CENTER ATTN: LES ANDERSON 271 CATALINA BLVD, CODE 413 SAN DIEGO CA 92151	1
GEORGE MASON UNIVERSITY ATTN: SUSHIL JAJODIA ISSE DEPT FAIRFAX, VA 22030-4444	1
DIRNSA ATTN: MICHAEL R. WARE DOD, NSA/CSS (R23) FT. GEORGE G. MEADE MD 20755-6000	1
DR. JIM RICHARDSON 3660 TECHNOLOGY DRIVE MINNEAPOLIS, MN 55418	1
LOUISIANA STATE UNIVERSITY COMPUTER SCIENCE DEPT ATTN: DR. PETER CHEN 257 COATES HALL BATON ROUGE, LA 70803	1

INSTITUTE OF TECH DEPT OF COMP SCI ATTN: DR. JAIDEEP SRIVASTAVA 4-192 EE/CS 200 UNION ST SE MINNEAPOLIS, MN 55455	1
GTE/BBN ATTN: MAURICE M. MCNEIL 9655 GRANITE RIDGE DRIVE SUITE 245 SAN DIEGO, CA 92123	1
UNIVERSITY OF FLORIDA ATTN: DR. SHARMA CHAKRAVARTHY E470 CSE BUILDING GAINESVILLE, FL 32611-6125	1