Program Analysis and Tuning Tools for a Parallel Object Oriented Language: An Experiment with the TAU System.*

Allen Malony, Bernd Mohr
University of Oregon
{mohr,malony}@cs.uoregon.edu

Peter Beckman, Dennis Gannon
Indiana University
{beckman,gannon}@cs.indiana.edu

1 Introduction

Supercomputer programming environments have fallen far behind desktop and workstation systems in terms of the sophistication of the programming tools that are available to build new applications. Part of the reason for this is economic: the desktop market place is several orders of magnitude bigger than that of the supercomputing world. Another explanation is that the traditional supercomputer users are not accustomed to having good programming tools provided with high performance hardware. This is especially true of users of massively parallel processing (MPP) systems. Indeed, most users are delighted if there is a compiler that generates good code for a single processor.

However, the market for MPP supercomputer systems is undergoing a radical transformation from that of a captive of the international defense industry to a vital and energetic component of high-end computing in various commercial sectors. New users are demanding portable programming tools because they see their investment in software being far more important than the money they spend on hardware. Furthermore, because of their roots in desktop technology, they are demanding more than simple compiler environments and message passing libraries.

Unfortunately, there are major problems that must be overcome if we are to raise the level of parallel program software design to that of desktop and workstation tools. Specifically, we must learn more about the integration of programming environment technology and parallel algorithm design. There are several important issues here, including

*This research is supported by ARPA under Rome Labs contract AF 30602-92-C-0135 and Fort Huachuca contract ARMY DABT63-94-C-0029

- How do we integrate parallel computer performance tools with programming languages and systems so that one can design good algorithms without an over-emphasis on architectural detail?

- Can we build portable parallel program debuggers?

- How can the successful models of program development tools for software engineering tasks be extended to programming that incorporates explicit parallelism?

In this paper we examine and evaluate an experimental programming environment designed at the University of Oregon that address some of these issues. This system, called $\mathcal{T}$ (for Tuning and Analysis Utilities), is part of the pC++ programming system being distributed by a consortium consisting of Indiana University, University of Oregon and the University of Colorado. pC++ is a programming language based upon a *concurrent aggregate* extension to C++. The pC++ programming system has been ported to most of the major MPP platforms. It consists of a set of language preprocessors, runtime libraries, and tools for program execution, analysis and tuning. As an integrated program and performance analysis environment, $\mathcal{T}$ consists of six special graphical tools. The five below are discussed in this paper.

- *Cosy* is a tool that provides a point-and-click interface for controlling compilation, instrumentation, processor configuration, and program execution.

- *Classy* provides an interactive display of the class hierarchy graph of pC++ programs.

- *Cagey* is a program browser based on a static call graph display.

- *Fancy* provides a class browser and interactive text tool for exploring the application.

- *Racy* is a graphical tool for displaying the execution profiles generated by the pC++ performance analysis system.

We illustrate the use of these tools from the perspective of the design and evaluation of a single application in pC++: a bitonic sort module that is used as part of a large N-Body simulation of cosmological evolution[3]. In the sections that follow we will demonstrate how the tools were used to analyze this module. Section 3 gives a brief description of the algorithm. Sections 4 and 5 show how the static analysis tools illustrate the structure of the program. In Section 6 we will describe how the dynamic analysis tools expose a potential performance bug.

2 The Bitonic Sort Module

To illustrate the programming tools in the $\mathcal{T}$ system we will focus on the task of analyzing and tuning a single application based on a classical parallel algorithm: Batcher's *bitonic sort*. Sequentially sub-optimal, but a very parallelizable sorting algorithm, bitonic sort is based on the *bitonic merge*, a natural application of divide-and-conquer. 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

$$a_0, a_1, a_2, ..., a_m$$

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

The bitonic merge can be stated in its most compact form as follows

```
input: an array of items a[0:2**n-1] containing a  bitonic sequence.
merge(a, n, direction){
   k = 2**(n-1)
   foreach i in 0 to k-1 do
        if(((direction == increasing) && (a[i] > a[i+k]) ||
           ((direction == decreasing) && (a[i] < a[i+k])))
                      swap(a[i], a[i+k]);
   //now a[0:k-1] and a[k:2**n-1] are both bitonic and
   //for all i in [0:k-1] and j in [k:2**n-1]
   //         a(i) <= a(j) if direction = increasing and
   //         a(j) <= a(i) if direction = decreasing
   merge(a[0:k-1], n-1, direction);  merge(a[k:2**n-1], n-1, 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_i and a_{i+1} is a bitonic sequence of two items for each even i. If we merge 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) we have sorted sequences of length 8. This is illustrated in Figure 1 below where the vertical axis represents time steps going down the page.

A data parallel version of this algorithm is given by

```
sort(a, n){
     m = 2**n;                 // m = size of array
     for(i = 1; i < n; i++){  // merge(i) step
        k = 2**i;
        forall(s in 0:m-1) d[s] = (s/k) mod 2; // d is the direction
        for(j = k/2; j > 0; j = j/2){ // exchange(j) step
           forall(s in 0:m-1) d2[s] = (s/j) mod 2;
           forall(s in 0:m-1) where(d2[s]==1)  tmp[s] = a[s-j];
           forall(s in 0:m-1) where(d2[s]==0)  tmp[s] = a[s+j];
           forall(s in 0:m-1) where((d == d2) && (a > tmp)) a = tmp;
           forall(s in 0:m-1) where((d != d2) && (a <= tmp)) a = tmp;
           }
```

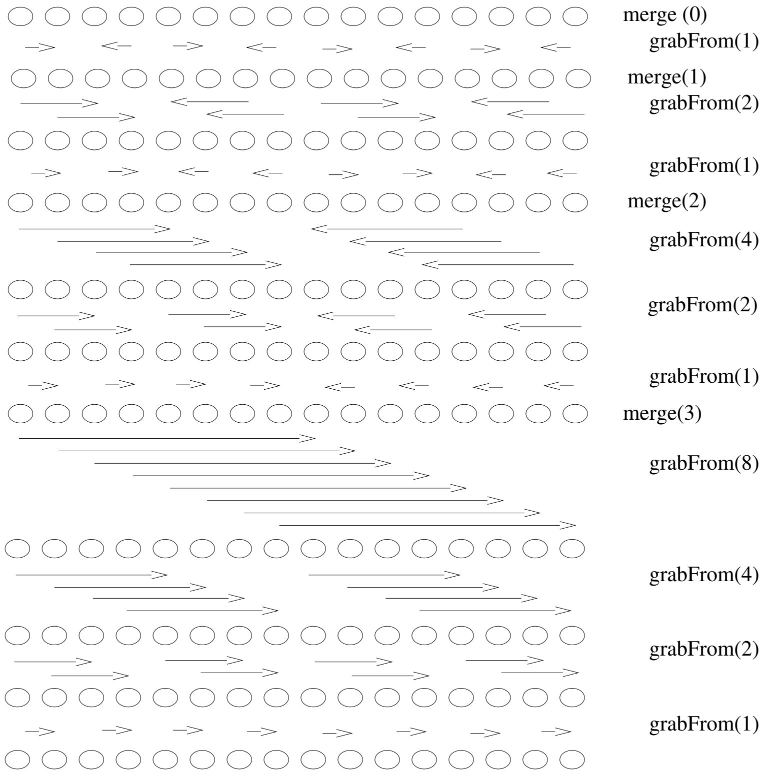

Figure 1: Data exchanges in the Bitonic Sort Algorithm

In an object parallel version of this algorithm we construct a set of objects e_i, $i = 0..n$ where each object contains a value from the sequence a_i and a temporary variable *temp*. The exchange step in the merge is replaced by communication where object i will *grab* a_{i+k} *from* e_{i+k} and store the value in its temp variable. In this operation, $k = 2^{n-1}$ if $i < 2^{n-1}$; $k = -2^{n-1}$ otherwise. Once the values have been accessed and stored in the temporary variable, we make a comparison and test. Object e_i will replace a_i with the *temp* value if the appropriate conditions are satisfied. The code for this can be derived from the data parallel version of the algorithm by rewriting it in terms of the action on a single array element as described above. As a C++ class this would appear as follows

```
class ListItem{
  public:
     item a;          // the data item
     item temp;       // a temporary value for the data swap step.
     int direction;  // a flag that store the direction of the current merge
     int i;           // the position of this object in the aggregate
     ListItem *thisCollection;  // a pointer to the collection
     void merge(distance);     // the bitonic merge
     void sort(n);             // the bitonic sort
     void grabFrom(distance);  // access the data from another ListItem
     localMerge(dist);   // the test and swap step
  };
```

```
ListItem::sort(n){
    for(d = 1; d < n; d++)  merge(d);
   }
ListItem::merge(int level){
    direction = (i/(2**level) % 2;
    for(int d = level-1; d >= 0; d--){
           int distance = 2**d;
           grabFrom(distance);
           localMerge(distance);
           barrier_sync(thisCollection);
         }
}
item::grabFrom(dist){
   if((i/dist) % 2) temp_data = thisAggregate[myPosition+dist].data;
   else             temp_data = thisAggregate[myPosition-dist].data;
}
ListItem::localMerge(int dist){
   int d2;
   d2 = (i/dist) % 2;
   if((direction == d2) && (a > temp)) a = temp;
   if((direction != d2) &&( a <= temp)) a = temp;
}
```

From the perspective of pC++ parallel programming, there is one very critical point in the code above. That is, there is a potential race condition in the merge step. To see this, observe that if each object in the collection is executing the merge function then, in the loop, one must make sure that the data read operation in `grabFrom` executes after the other object has completed the `localMerge` step of the previous iteration. To enforce this rule we have inserted a barrier synchronization step which causes all object to wait at this position before advancing to the next iteration. The insertion of this operation has serious impact on the asymptotic complexity of this algorithm. If there are 2^n processors, the standard algorithm has time complexity $O(n^2)$. By adding the barrier synchronization, we have increased the complexity to $O(n^3)$.

Another serious problem with the algorithm is that it is very inefficient if the number of processors available is less than 2^n. In the N-body application, we need to sort tens of millions of particles on machines ranging from 8 to 1024 processors. Consequently, we need a version of the algorithm that requires fewer processors. Fortunately, there is an easy way to increase the granularity of the computation so that each object holds a vector of particles. If we replace the `ListItem` class above with a class called

```
class Pvector{
   item a[K];   // K is the size of the local vector
   item temp[K];
   ...
 };
```

and a standard quicksort function `qsort(direction)` to the class that sorts the local vector of item in the given direction, then the changes to the program are very small. We only need to make sure that the initial vectors are sorted and that they are resorted at the end of each merge step.

```
Pvector::merge(level){
    direction = (i/(2**level)) % 2;
    for(int d = level-1; d >= 0; d--){
           int distance = 2**d;
           grabFrom(distance);
           localMerge(distance);
           barrier_sync(thisCollection);
         }
    qsort(direction);
}
Pvector::sort(n){
    qsort(i%2);
    barrier_sync(thisCollection);
    for(d = 1; d < n; d++)  merge(d);
   }
```

In addition to the changes above, the function `localMerge` must be modified to operate on a vector of data. If we assume that the quicksort computation runs with an average execution time of $DKlog(K)$ for some constant D and if we ignore the cost of the barrier synchronization, and assume that there are $P = 2^n$ processors available and the size of the list to sort is N, then the time to sort is roughly

$$T(N) = \frac{N}{P} log(P)(Clog(P) + Dlog(N))$$

where C is a constant that depends upon communication speed. Given a sequential complexity of $DNlog(N)$ we see that the parallel speed-up is of the form

$$Speedup(N,P) = \frac{\frac{P}{log(P)}}{1 + \frac{Clog(P)}{Dlog(N)}}$$

which, for large N, is of the form $\frac{P}{log(P)}$.

It is important to notice that at the end of each merge step, just before the quicksort is called, the local vector is a bitonic sequence due to its construction. Hence, it is possible to apply a sequential bitonic merge that runs in time $O(K)$ rather than the $O(Klog(K))$ quicksort. This changes the approximate execution time to be

$$T_{modified}(N) = \frac{N}{P} Clog^2(P) + D\frac{N}{P} log(\frac{N}{P}))$$

and the speed-up becomes

$$Speedup(N,P) = \frac{P}{1 + \frac{Clog(P)}{Dlog(N)}}$$

which, for large N, is of the form P.

Our focus for the remainder of this paper will be on using the $\mathcal{T}$ tools to explore several questions about this algorithm posed by the analysis above. In particular, we are interested in using the tools to consider the following.

1. Does barrier synchronization adds a significant penalty to the complexity that is not accounted for in the analysis above? What percent of the time is actually spent in barrier synchronization for problems of a size that is significant for the application?

2. Given that bitonic merge is communication intensive (in fact, the bisection bandwidth of the communication graph scales linearly with the problem size), what percent of the time is spent in communication on different machines?

3. How significant is the improvement in the "modified" version of the algorithm which replaces the quicksort in each merge step by a local bitonic merge?

3 The pC++ Program Analysis Environment

In this section, we give a brief overview of $\mathcal{T}$, a first prototype for an integrated portable pC++ program and performance analysis environment.

Elements of the $\mathcal{T}$ graphical interface represent objects of the pC++ programming model: collections, classes, methods, and functions. These language-level objects appear in all $\mathcal{T}$ tools. By plan, $\mathcal{T}$ was designed and developed in concert with the pC++ language system. It leverages pC++ language technology, especially in its use of the Sage++ toolkit [2] as an interface to the pC++ compiler for instrumentation and for accessing properties of program objects. $\mathcal{T}$ is also integrated with the pC++ runtime system for profiling and tracing support. Because pC++ is intended to be portable, the tools are built to be portable as well. C++ and C are used to ensure portable and efficient implementation, and similar reasons led us to choose Tcl/Tk [4, 5] for the graphical interface.

The $\mathcal{T}$ tools are implemented as graphical *hypertools*. While the tools are distinct, providing unique capabilities, they can act in combination to provide enhanced functionality. 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). With this design approach, the toolset can be easily extended. $\mathcal{T}$ can also be retargeted to other programming environments, for instance those based on Fortran, which Sage++ toolkit supports.

One important goal in $\mathcal{T}$'s development was to make the toolset as user-friendly as possible. For this purpose, many elements of the graphical user interface are analogous to *links* in *hypertext* systems: clicking on them brings up windows which describe the element in more detail. This allows the user to explore properties of the application by simply interacting with elements of most interest. The $\mathcal{T}$ tools also support the concept of *global features*. If a global feature is invoked in any of the tools, it is automatically executed in all currently running tools. Examples of global features include `select-function`, `select-class`, and `switch-application`. For good measure, $\mathcal{T}$ comes with a full hypertext help system.

Figure 2 shows the pC++ programming environment and the associated $\mathcal{T}$ tools architecture. The pC++ compiler frontend takes a user program and pC++ class library definitions (which provide predefined collection types) and parses them into an *abstract syntax* *tree* (AST). All access to the AST is done via the Sage++ library. Through command line switches, the user can choose to compile a program for profiling, tracing, and breakpoint debugging. In these cases, the instrumentor is invoked to do the necessary instrumentation in the AST. The pC++ backend transforms the AST into plain C++ with calls into the pC++ runtime system which supports both shared and distributed memory platforms. This C++ source code is then compiled and linked by the C++ compiler on the target system. The compilation and execution of pC++ programs can be controlled by *cosy* (**CO**mpile manager **S**tatus displa**Y**). This tool provides a high-level graphical interface for setting compilation and execution parameters and selecting the parallel machine where a program will run.

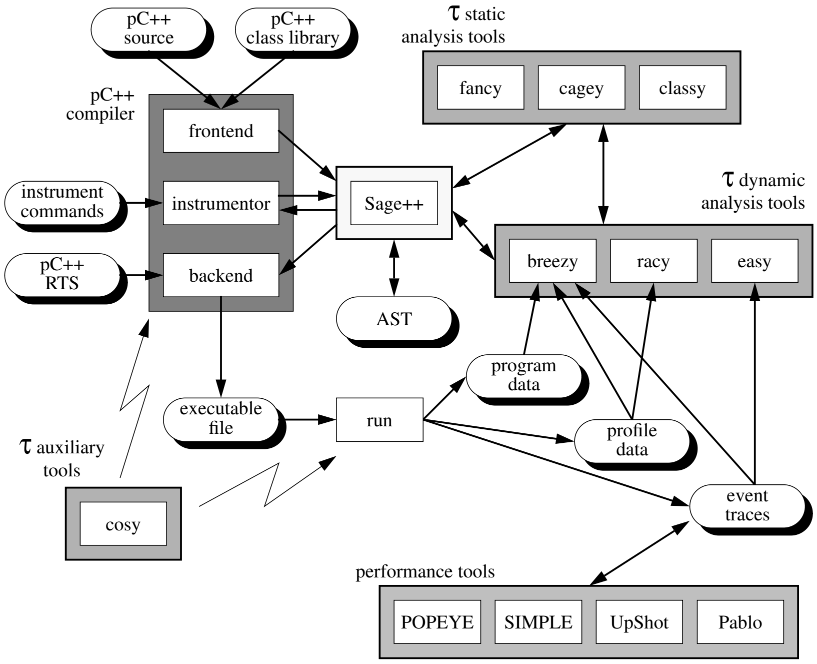

Figure 2: pC++ Programming Environment and $\mathcal{T}$ Tools Architecture

The program and performance analysis environment is shown on the right side of Figure 2. They include the integrated $\mathcal{T}$ tools, profiling and tracing support, and interfaces to stand-alone performance analysis tools developed partly by other groups [6, 7, 8, 9]. The $\mathcal{T}$ toolset provides support for accessing static information about the program and for querying and analyzing dynamic data obtained from program execution. A more detailed discussion of the $\mathcal{T}$ tools can be found in [10, 11, 12].

4 A Tour of the Sort Program: Static Analysis

The pC++ implementation of the Sort program is easily derived from the class object defined above. A pC++ *collection* of `Pvector` objects is created. The member functions of the `Pvector` class are divided into two categories: those functions that describe operations that are purely local to the object, and those functions that involve global operations on the collection or interaction with other elements of the collection. The second category of

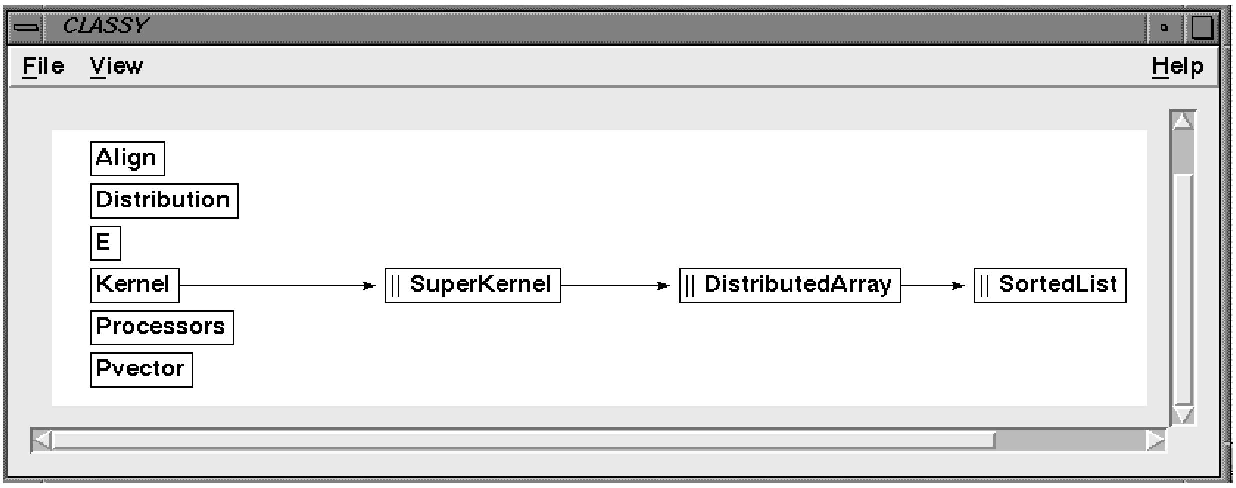

Figure 3: Class Inheritance Hierarchy Graph Display

functions are moved to the collection structure and are called *MethodOfElement* members. Our collection is derived from the library collection *DistributedArray*.

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

Figure 4 illustrates the view of our *SortedList* collection. From this *fancy* panel view, we can select any member function (with a mouse click) and get a text display of that function (e.g., `SortedList`). In addition to examining the code, the user can examine many other static properties of the program. For example, the inheritance graph for the classes and collections in the pC++ application are illustrated in Figure 3. Another display is the static call graph of our program. It is automatically generated and displayed with the *cagey* tool as illustrated in Figure 5. This view of the program is completely adjustable: the use can select the detail and depth of the displayed information by unfolding and folding selected function calls.

5 Compiling and Executing the Program: COSY

In our experiments, we need to run the pC++ program in a number of different configurations on different machines. These include several different levels of instrumentation (none, complete, and selective). In addition, we need to run the program with different numbers of processors on different machines. Keeping track of all the required compile and runtime flags necessary to accomplish our goal is a non-trivial task. The *cosy* tool provides a very simple set of menus for configuring and running a pC++ program. In our case, we restricted profiling to the quick-sort routine `Qsort`, the main bitonic sort `sort`, the bitonic merge `merge`, the routine that contains all the communications `grabFrom`, and the barrier synchronization code `pcxx_Barrier`. While there are many more small functions that are executed, we found

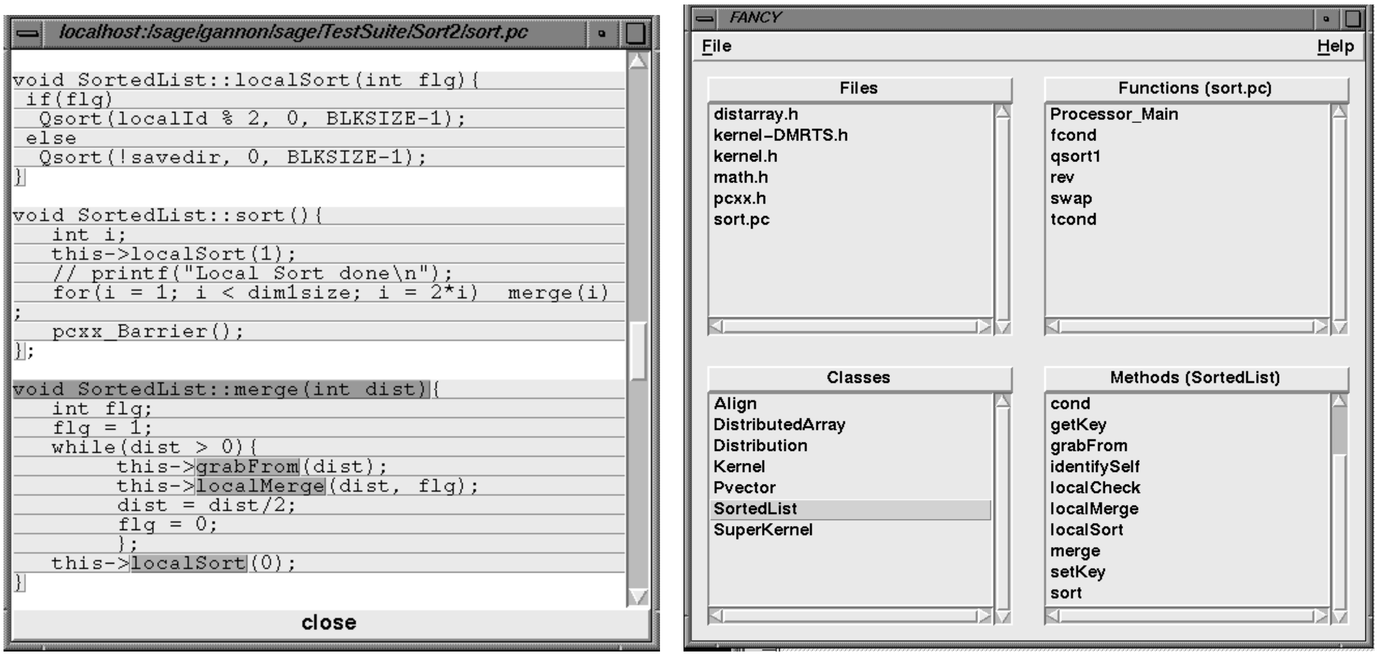

Figure 4: Class Browser and Code View Tool

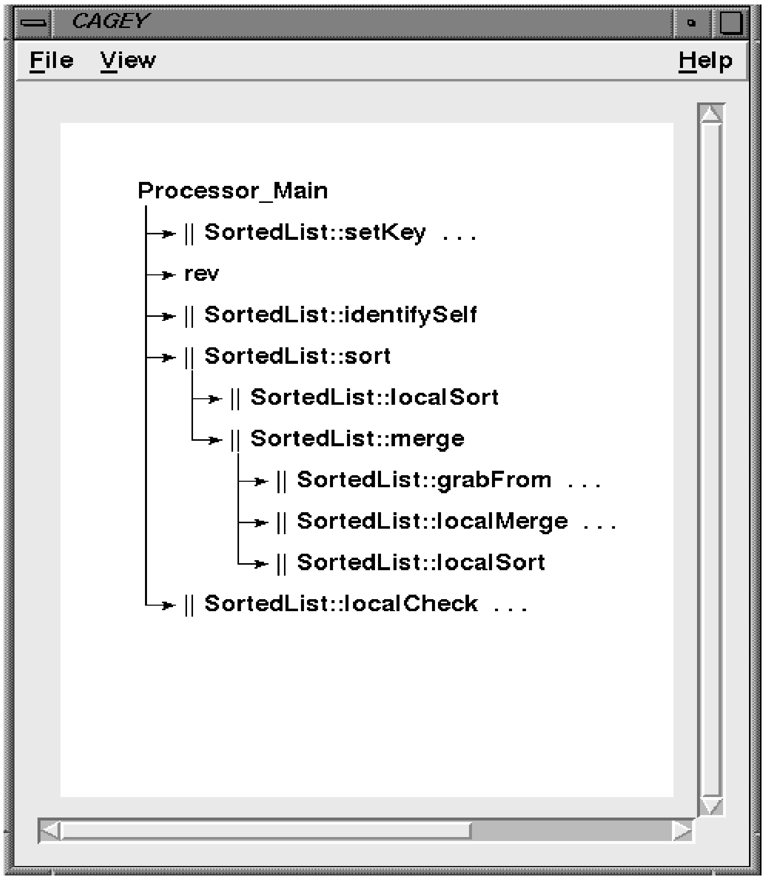

Figure 5: Static Call Graph Window

when we included them all, the size of the event trace file was too large and the execution time were severely distorted. However, when we used this restricted set of functions, the trace files were very compact and the impact of instrumentation on total execution time was less than 5 percent.

6 Sort Program Execution: Dynamic Analysis

Dynamic program analysis tools allow the user to explore and analyze program execution behavior. The most critical factor for the user during performance debugging is how the high-level program semantics are related to the measurement results. $\mathcal{T}$ helps in presenting the results in terms of pC++ language objects and in supporting global features that allow the user to quickly locate the corresponding routine in the callgraph or source text by simply clicking on the related measurement result or state objects.

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

The $\mathcal{T}$ execution profile tool *racy* provides a very simple and portable means of understanding where a parallel program spends its time. The initial display is a set of bar graphs for each processor showing the percent of the execution time spent in each of the profiled routines in the program. To study the questions posed at the end of section 2, we have run the program on a data set of 1024 `Pvector` element objects each containing a list of 1024 items; a total of 2^{20} objects to sort. We ran this on 16 nodes of a Silicon Graphics Challenge machine, and on 16 and 64 nodes of an Intel Paragon. Recall that pC++ is completely portable, so all that was required to compile for these two environments was to use the *cosy* tool to select the appropriate system name; no source code needed to be modified.

The results for the 16 node SGI and the 64 node Paragon are illustrated in Figures 6 and 7. There are three displays in each figure. The processor profile bar graphs are shown in the lower right. By using the mouse we selected the bar graph that describes the *mean* profile shown at the top. The result is given in the upper right display. This version of the bar graph lists the names of each function and gives the numerical percentage values of execution time. Because we are also interested in the time spent in the barrier synchronization we have asked *racy* to plot that for all processors. This data is shown by the chart on the left.

There are several interesting observations that we can make about our sorting program from these displays. First it is clear that the time spent in the barrier synchronization routine is not large. It is about 5 percent for the SGI and less that 2 percent for the Paragon. One interesting observation about the Paragon numbers are that high-numbered processors spend less time in the barrier than do low numbered processors. The reason for this is uncertain. Every aspect of the algorithm design is processor independent and the runtime system design is relatively symmetric with respect to processors (i.e. the barrier algorithm is based on a tree reduction). For the SGI, the time spent in the barrier is more random, relating to the fact that processors are occasionally interrupted to carry out other tasks as requested by the operating system.

The reader may notice that there is one difference between the SGI and the Paragon displays. In the Paragon version there is an extra function that is profiled that is represented by a white bar and is unlabeled in the upper right display. This function is the pC++ *poll loop* which supports the global name space on the distributed memory Paragon. This is the third largest consumer of time at 9.74%. In a distributed memory machine based on message passing, when a processor wishes to ask for data from another processor it must send a message. Because the Paragon does not support an active message interrupt, it is necessary to have each processor periodically poll for incoming request and service them. In this case the profile information tells us that each processor spends more time polling for incoming requests than it does in asking for data (in the `grabFrom` function). Hence, it may be possible to "fine tune" the runtime system so that it polls less often.

What have we learned about our algorithm? First, the barrier synchronization costs are relatively small, even though the asymptotic analysis suggests that they may be a problem. Second, we see that the largest cost is the time spent in the local quicksort (72% of the time on the SGI and 51% of the time on the Paragon). This observation leads us to consider the third question posed at the end of section 2: How significantly can the algorithm be improved by replacing quicksort at each merge step with a local bitonic merge?

After making the experiments described above, we reprogrammed the algorithm so that the call to the quicksort at the end of each merge step is replaced by a call to a local bitonic merge operation. The difference in performance was impressive. The execution time of the original algorithm on 64 processors it was 30.13 seconds. After the change to the merge function, this time dropped to 3.56 seconds. On the SGI, the original version of the sort ran in time 47.4 seconds with 16 processors. After the algorithm change it ran in 5.1 seconds. Again, from using *racy* to plot the mean profiles shown in Figures 8 it is clear that the new algorithm is no more balanced than before and only improvements in interprocessor communications will make a significant difference in overall performance.

7 Conclusion

The evolving requirements for the use of high-performance parallel computing systems are presenting new challenges for programming environment technology. The $\mathcal{T}$ environment is representative of a new class of collaborative program and performance tools that, fundamentally, are integrated into a parallel programming system and support portable, high-level program observation, analysis, and interaction. In this paper, we demonstrated how the $\mathcal{T}$ tools can be used to explore the static program structure (class hierarchy, source code, and call graphs) and how the performance profiling tools can answer questions about dynamic system and algorithm behavior. In particular, we saw the importance of coupled program and performance analysis wherein serious bottlenecks in a code can be linked back to high-level algorithm design choices, as in the case of bitonic sort where a replacement of quicksort with bitonic merge at each merge step increased performance by nearly an order of magnitude.

Although the experience with $\mathcal{T}$ has been exceedingly positive, there are still several important directions we want to pursue. First, there is the obvious need for a parallel debugging system that provides high-level access to program objects. Hence, much of our current work focuses on *breezy*, a parallel barrier breakpoint debugger that supports runtime

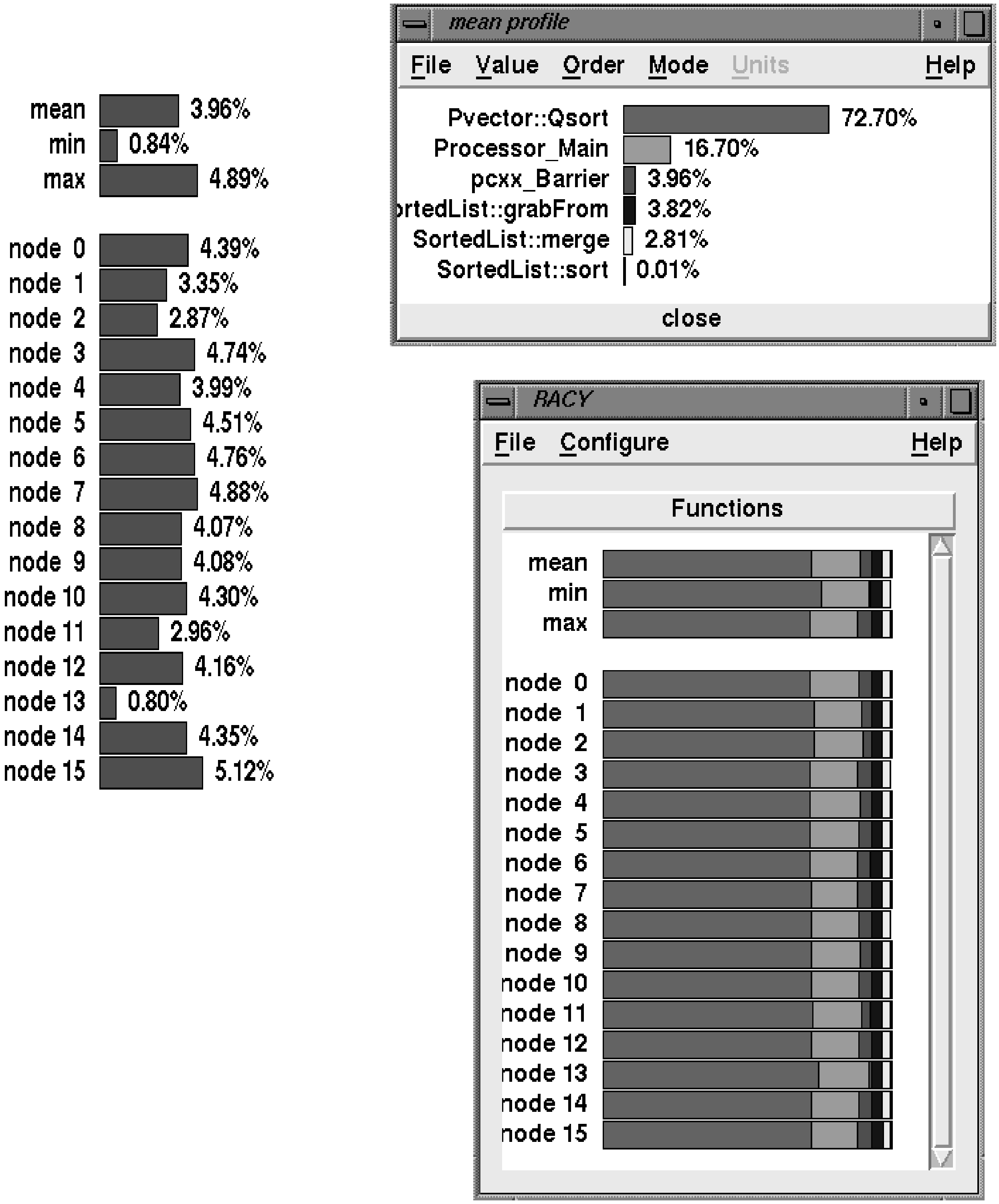

Figure 6: Performance Profiles from Racy on 16 node SGI Challenge

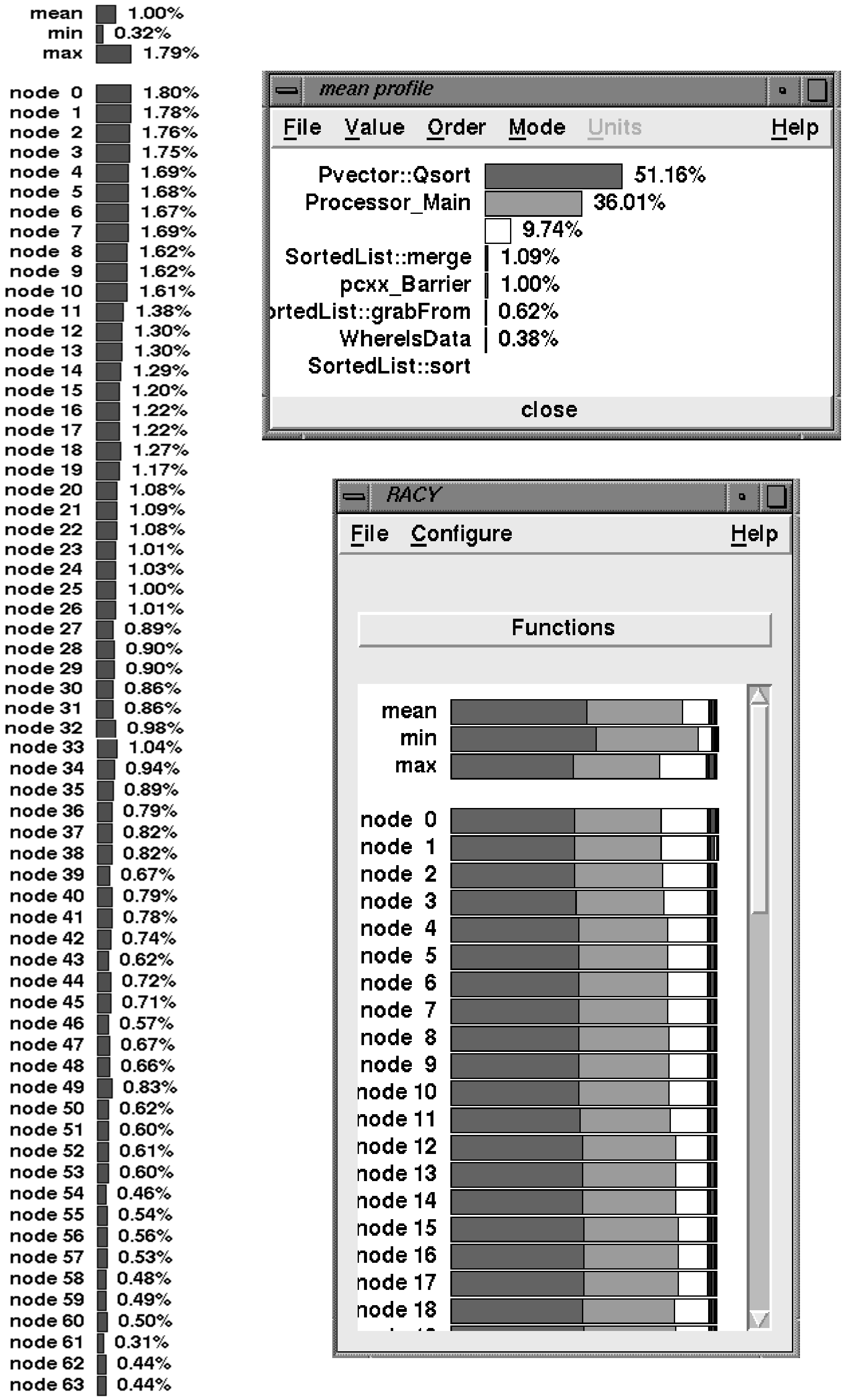

Figure 7: Performance Profiles from Racy on 64 node Paragon

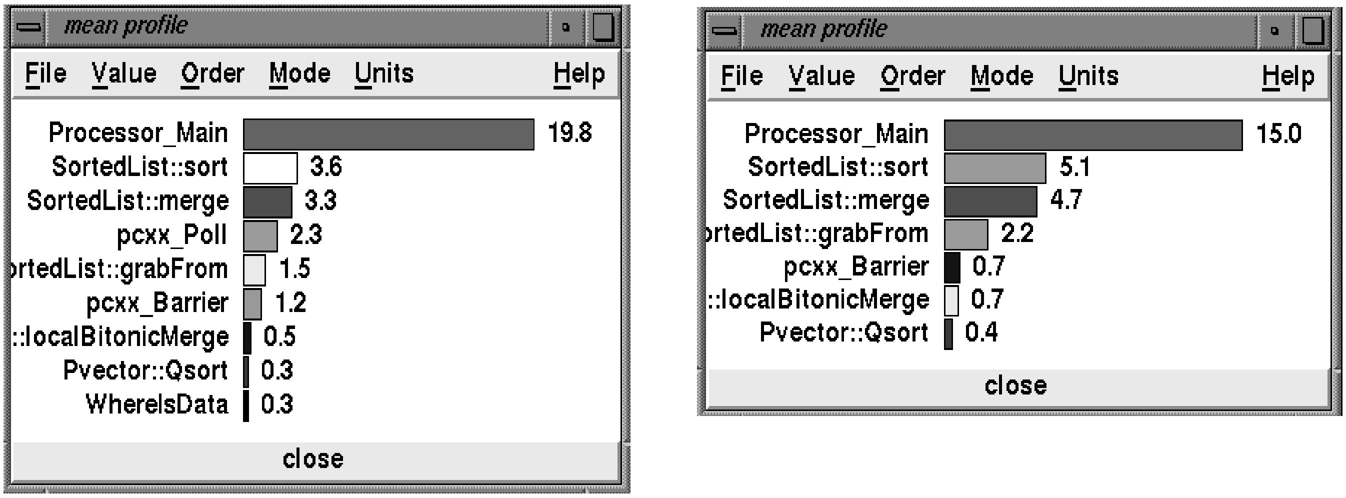

Figure 8: The modified algorithm. On the left, the mean profile showing total time spent in each function including the functions it calls for 64 nodes on the Intel Paragon. On the right, the same graph but for 16 nodes of the SGI Challenge.

access to pC++ program state. The *breezy* tool actually implements a robust infrastructure for program access upon which several external runtime tools can be built. A second direction is towards more expanded support for performance experimentation and diagnosis, particularly for scalability analysis. We are attempting to integrate a simulation-based performance extrapolation tool and a semi-automated performance diagnosis system [13] into the $\mathcal{T}$ toolset. In addition, we are working on better ways to maintain a database of program measurements that record the progress of a sequence of experiments and to display comparative performance behaviors across multiple executions for different numbers of processors. Finally, we are improving the design of the $\mathcal{T}$ tools to facilitate the transfer of the technology to other language environments, in particular HPF.

For more information ...

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

References

[1] D. Knuth, *The art of computer programming*, Vol. 3, Searching and Sorting, Reading, Mass., Addison-Wesley Pub. Co., 1968.

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

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

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

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

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

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

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

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

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

[11] B. Mohr, D. Brown, A. Malony, *TAU: A Portable Parallel Program Analysis Environment for pC++* , Proc. of CONPAR 94 - VAPP VI, Linz, Austria, Springer Verlag, LNCS 854, pp. 29–40, Sept. 1994.

[12] D. Brown, S. Hackstadt, A. Malony, B. Mohr, *Program Analysis Environments for Parallel Language Systems: The TAU Environment*, Proc. of the Workshop on Environments and Tools For Parallel Scientific Computing, Townsend, Tennessee, pp. 162–171, May 1994.

[13] B. Robert Helm, Allen D. Malony, Steve Fickas, *Capturing and Automating Performance Diagnosis: The Poirot Approach*, to be published in Proc. 9th Inter. Parallel Processing Symp. (IPPS '95), Santa Barbara, California, April 1995.