Performance Tools for Today's HPC: Are We Addressing the Right Issues?

Cherri M. Pancake
Northwest Alliance for Computational Science and Engineering
Oregon State University
pancake@nacse.org

Abstract

HPC application developers can no longer afford the luxury of programming to just one platform. The usefulness and longevity of software now depend on portability as well as performance. This paper examines the appropriateness of existing tools in developing and tuning applications that must be both efficient and portable. A series of design tradeoffs have been made, with the result that today's tools are slanted toward one goal or the other but don't respond adequately to both. These tradeoffs are explored through a series of examples drawn from current tools. Suggestions are presented for how tools might better support the evolving needs of HPC programmers.

The past ten years have borne witness to a remarkable evolution in high-performance computing (HPC) architectures. As recently as 1993, the norm was still a tightly-coupled collection of custom-built, homogeneous processors with on-board memory. Today, the system an HPC programmer deals with is likely to be a loosely coupled or distributed aggregate of commodity components, where individual nodes may be heterogeneous and may, in turn, include heterogeneous processors and multiple levels of memory. Programming models, too, have changed -- although not as much, perhaps, as they ought. Where message-passing was the single dominant paradigm in the early 1990s, today's HPC applications are likely to combine message-passing modules with code that uses shared-memory operations and library routines employing low-level thread primitives.

As a result, the concept of a parallel application exhibiting good performance characteristics has become almost an oxymoron. A decade ago, HPC programming was a difficult task [10], requiring a great deal of human effort and a healthy dose of insight in order to achieve efficiency rates on the order of 20-50%. The achievable percentage has dropped steadily over recent years [8], and on today's machines is more likely to be 2-5%. Yet performance remains a key goal for most HPC application developers [9], and it is interesting to speculate just how bad efficiency might be if programmers were not making efforts to achieve it.

A second result of the evolutionary path of HPC systems is that code portability has become of critical importance. Programming for a single platform is no longer practical. The typical hardware lifespan is often less than the software development cycle, meaning that a program may never actually run on the machine where it was developed. For a variety of sociopolitical reasons, there is also an increasing need to run parallel applications at multiple sites or across computational grids, where different system configurations and perhaps very different architectures are in use. Similarly, there is increasing pressure on HPC developers to extend the useful lifetime of software by re-using components from previous applications and by linking applications to form complex aggregates (e.g., linking hydrologic and atmospheric models to study global cycles of carbon dioxide). Code portability is clearly important in addressing all of these issues.

The need for portability affects the application development process in profound ways. The programmer must now deal with multiple target machines and as users point out, these targets change constantly [2]. It is not just a matter of adjusting the application to new machine architectures. Each revision of the application development environment (i.e., each new release of the operating system, compiler, system libraries, etc.) can expose new bugs or incompatibilities, even when the application itself has not been altered. Since these revisions tend to be very frequent in HPC environments, a portable code must constantly be re-tested and re-evaluated -- in effect, it is always under revision [2, 8].

The situation has been complicated by the emergence of heterogeneous applications. In some cases, it is perfectly natural for an application to span multiple platforms, since components may have very different computational requirements (e.g., a computational model that receives input from a signal-processing stream and channels output to a volume-rendering component). In others, variations in the availability of distributed computing resources -- due to differences in peak-hour schedules, institutional priorities, etc. -- have precipitated an interest in exploiting other groups' resources by means of computational grids.

Platform variability applies to other aspects of the program development environment as well. It is no longer the case that programmers log directly into a target HPC machine. Typically, the user accesses a desktop system, which in turn communicates with the HPC machine or, increasingly, an intermediate host machine. The user may never interact directly with the target system at all. Instead, he or she may develop and test code on other systems and later submit the application in the form of batch jobs that will be scheduled for execution on the target. New possibilities for error or incompatibility may be introduced at each level; the user might not even be aware of the differences from one level to the next.

Given this context, what should performance tools do to facilitate the application developer's task? This paper outlines four key characteristics of today's HPC applications and discusses the extent to which performance tools support or hinder them. It will be seen that a series of design tradeoffs have been made, with the result that tools are slanted toward efficiency or portability, but don't respond adequately to both. In each case, suggestions are presented for how tool support might be improved to better meet the evolving needs of HPC programmers.

Supporting Code Portability

Perhaps the most obvious characteristic of current HPC applications is that they **need to run on many different computing platforms, some of which may be heterogeneous and/or distributed**. This is the essence of the code portability problem: the need to execute correctly despite variation in the underlying architecture, operating system, or software libraries.

There is no doubt that this challenges the capabilities of performance tools. They are responsible for acquiring and analyzing data from hardware and software counters, event logs, and other run-time sources. In the absence of standards for what data are to be collected, or how they are to be represented, tools are limited in their ability to generalize results across different machines. Unfortunately, tool developers have made the tradeoff that although a given tool may run on multiple HPC platforms, support for code portability is incomplete. There are two aspects to this problem.

First, it is the tool back-end that has been re-targeted, not the front-end. Tools suck as Vampir

[6], DEEP [1], and Jumpshot [15] have all been implemented for the most common target HPC hosts. With one exception, however, their front-ends must execute on UNIX workstations. (In fact, many tools interfaces are not even tested rigorously across the full range of UNIX and LINUX variants.) This does not respond to the exigencies of most user sites, where Windows and Macintosh desktop systems are also commonplace. Given the widespread availability of Web browsers with plug-in capabilities, the problem can be solved, but to date only Jumpshot [15] has attempted to do so.

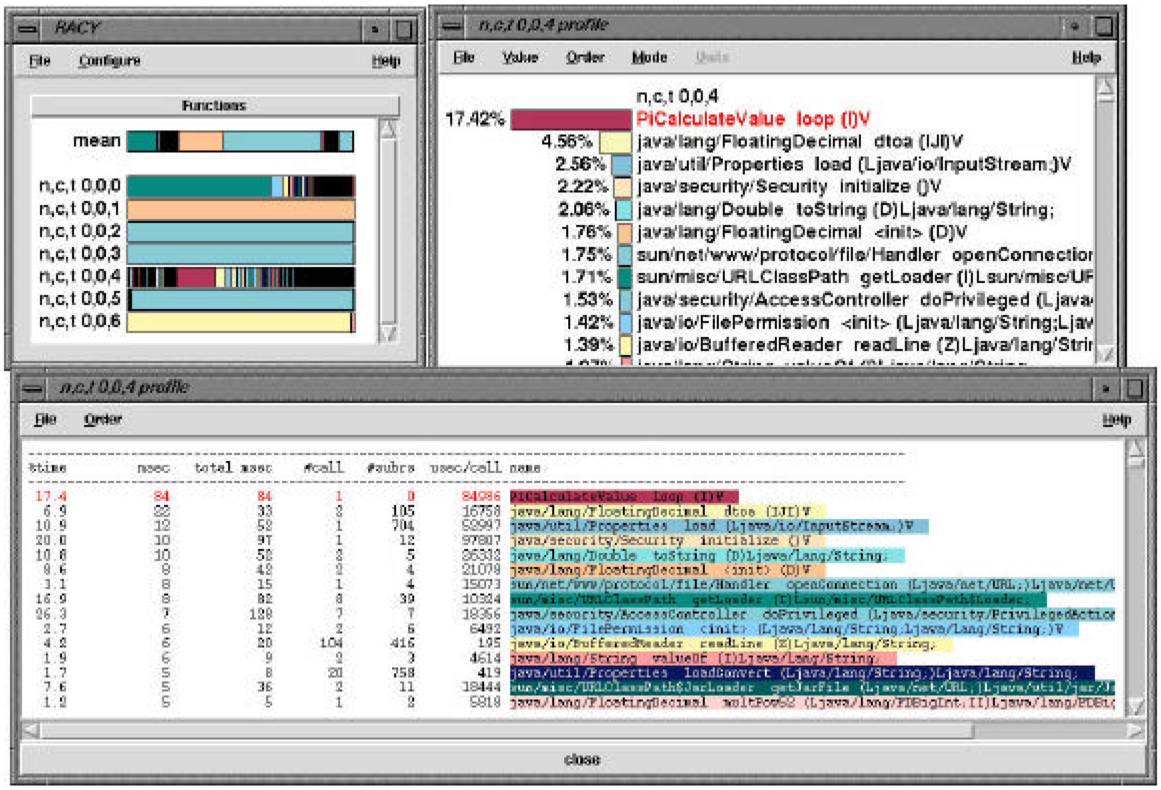
Second, current tools are not capable of running *across* multiple platforms. This means that even if a tool has been ported to different architectures or operating environments, it is not possible to apply the tool to analyze the performance of a heterogeneous application -- even when the heterogeneity is restricted to platforms from a single vendor. At best, the performance of heterogeneous components must be analyzed on an individual basis. For computational grid applications, which may be assigned to arbitrary computers at each execution, no tools are currently available.

To summarize, tools may support portability by being available for more than one target platform, but they do not yet accommodate variability in display (desktop) platforms, nor do they support applications that execute across multiple target platforms.

Supporting Performance Variability

Frese and Peterkin [2] proposed a rule-of-thumb for estimating how long a portable code will continue functioning: until the new processor boards are installed *or* the parallel environment changes *or* the shared library changes *or* the next system upgrade *or* the next system reconfiguration *or* the next reboot *or* the system load changes. While their list is somewhat tongue-in-cheek, they are accurate in pointing out that **portable code tends to expose performance quirks**. Even a small change in the operating environment may result in performance degradation, if not failure, or a portable program. The dilemma for the programmer, as they point out, is that "correcting for platform-specific quirks lessens portability."

Current tools provide few mechanisms for addressing this problem. Most provide ordered lists of "top" resource-using subprocedures; Figure 1 shows an example where routines have been ordered to reflect how much CPU time each uses. This is not necessarily the best way to identify where performance improvement efforts should be concentrated, however, since the simple consumption of CPU or other resources does not mean that there is a corresponding potential for improvement.



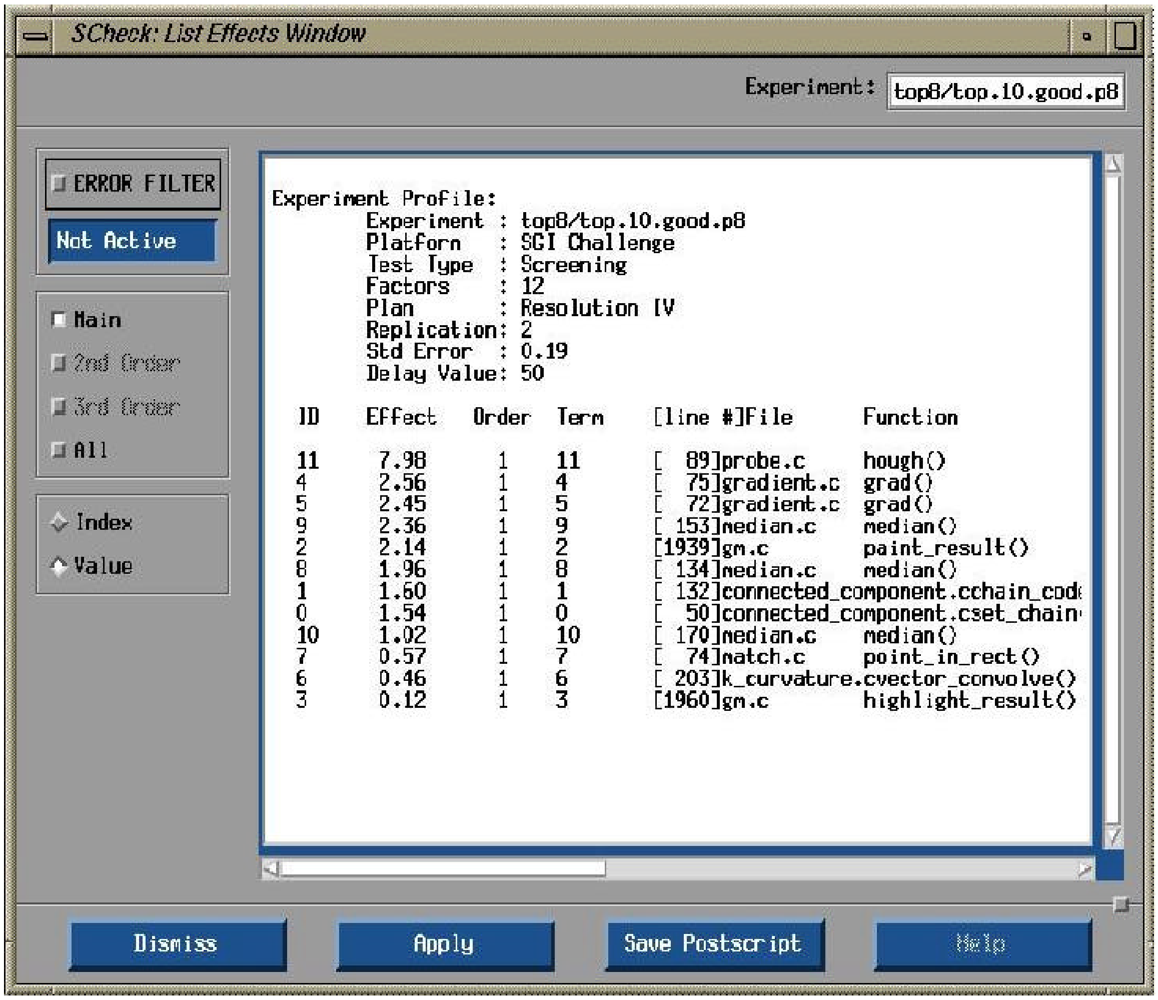
```
n,c,t 0,0,4
17.42%  PiCalculateValue  loop (I)V
 4.56%  java/lang/FloatingDecimal  dtoa (IJI)V
 2.56%  java/util/Properties  load (Ljava/io/InputStream;)V
 2.22%  java/security/Security  initialize ()V
 2.06%  java/lang/Double  toString (D)Ljava/lang/String;
 1.76%  java/lang/FloatingDecimal  <init> (D)V
 1.75%  sun/net/www/protocol/file/Handler  openConnectior
 1.71%  sun/misc/URLClassPath  getLoader (I)Lsun/misc/UF
 1.53%  java/security/AccessController  doPrivileged (Ljava
 1.42%  java/io/FilePermission  <init> (Ljava/lang/String;Ljav
 1.39%  java/io/BufferedReader  readLine (Z)Ljava/lang/Strir
```

```
-------------------------------------------------------------------------------------
%time         msec   total msec     #call   #subrs  usec/call name
-------------------------------------------------------------------------------------
 17.4           84           84         1        0      84086 PiCalculateValue  loop (I)V
  6.9           22           33         2      105      16758 java/lang/FloatingDecimal  dtoa (IJI)V
 10.9           12           52         1      704      52997 java/util/Properties  load (Ljava/io/InputStream;)V
 20.0           10           97         1       12      97807 java/security/Security  initialize ()V
 10.8           10           52         2        5      26332 java/lang/Double  toString (D)Ljava/lang/String;
  8.6            8           42         2        4      21078 java/lang/FloatingDecimal  <init> (D)V
  3.1            8           15         1        4      15073 sun/net/www/protocol/file/Handler  openConnection (Ljava/net/URL;)Ljava/net/U
 16.9            8           82         8       39      10324 sun/misc/URLClassPath  getLoader (I)Lsun/misc/URLClassPath$Loader;
 26.3            7          128         7        7      18356 java/security/AccessController  doPrivileged (Ljava/security/PrivilegedActior
  2.7            6           12         2        6       6492 java/io/FilePermission  <init> (Ljava/lang/String;Ljava/lang/String;)V
  4.2            6           20       104      416        195 java/io/BufferedReader  readLine (Z)Ljava/lang/String;
  1.9            6            9         2        3       4614 java/lang/String  valueOf (I)Ljava/lang/String;
  1.7            5            8        20      758        419 java/util/Properties  loadConvert (Ljava/lang/String;)Ljava/lang/String;
  7.6            5           36         2       11      18444 sun/misc/URLClassPath$JarLoader  getJarFile (Ljava/net/URL;)Ljava/util/jar/J
  1.2            5            5         1        2       5818 java/lang/FloatingDecimal  multPow52 (Ljava/lang/FDBigInt;II)Ljava/lang/FDBig
```

Figure 1. Tau [5] displays; upper right ranks the top CPU-consuming routines.

The problem is that while current tools present summaries of the raw performance data gathered during program execution, they do little or no analysis of those data. What is needed are more sophisticated analysis capabilities that can provide some indication of where user efforts are most likely to pay off. Two recent research efforts demonstrate the type of analysis that is possible given current data-collecting mechanisms.

S-check [12] applies classic sensitivity analysis techniques from statistics to predict how improvements in one part of a program will affect overall performance. The tool then orders the code units, not according to their resource consumption, but by their potential contribution to improvement. Moreover, the authors demonstrate that when this technique is applied to executions on different platforms, the results can be compared to identify where changes will benefit both performances.



Experiment: top8/top.10.good.p8

```
Experiment Profile:
        Experiment : top8/top.10.good.p8
        Platform   : SGI Challenge
        Test Type  : Screening
        Factors    : 12
        Plan       : Resolution IV
        Replication: 2
        Std Error  : 0.19
        Delay Value: 50

  ID     Effect   Order   Term     [line #]File       Function

  11      7.98      1      11      [  89]probe.c      hough()
  4       2.56      1      4       [  75]gradient.c   grad()
  5       2.45      1      5       [  72]gradient.c   grad()
  9       2.36      1      9       [ 153]median.c     median()
  2       2.14      1      2       [1939]gm.c         paint_result()
  8       1.96      1      8       [ 134]median.c     median()
  1       1.60      1      1       [ 132]connected_component.cchain_code
  0       1.54      1      0       [  50]connected_component.cset_chain
  10      1.02      1      10      [ 170]median.c     median()
  7       0.57      1      7       [  74]match.c      point_in_rect()
  6       0.46      1      6       [ 203]k_curvature.cvector_convolve()
  3       0.12      1      3       [1960]gm.c         highlight_result()
```

Figure 2. S-check [12] ranks code according to how much impact improvements will have on overall performance.

Merlin [4] also attempts to help the user identify the most promising approach, but in a very different way. Raw performance data is analyzed to identify performance symptoms. Using a database that encapsulates the knowledge gained by experienced performance analysts, the symptoms are then combined and mapped to diagnoses, which in turn are mapped to suggested solutions. The user is presented with the symptom (e.g., "poor speedup"), a list of likely diagnoses (e.g., "spreading overhead of this loop is too high"), descriptions of those diagnoses, and list of the program tuning steps (loop fusion, interchange, etc.) an expert would try.

These two research efforts exemplify what tools should be doing in order to help users cope with performance variability problems. Rather than simply presenting data summaries -- which is as far as current tools go -- a tool should be able to indicate where user efforts will yield the most impact, and perhaps even point the user to the remedial steps needed.

Supporting Code Complexity

A third characteristic with significant impact on tool usability is the fact that **portable applications are typically complex, involving a collection of semi-independent components**. Consider the example given by Frese and Peterkin [2], which is comparable to many others identified in our field studies of HPC programmers [10]. The magneto-hydrodynamics code they described is comprised of "a few tens of thousands of source lines and a few tens of include files." There are a few hundred subroutines, each stored in a separate source file that is dependent on some of the include files. A handful of customized component libraries are also used. The application is written partly in Fortran and partly in C.

This type of structure poses several problems for current tools, even apart from the issue of multiple source languages. Existing performance tools do not support very large-scale or complex source structures, balking not only at large numbers of source files, but sometimes even at multiple source code directories. As a result, users are often forced to analyze and tune their program components in isolation from one another. Even when the tool can accommodate the entire application, results are presented using displays that simply do not scale well to the massive quantities of data produced during execution. Figure 3 shows how quickly a display becomes cluttered. This screendump from AIMS [14] portrays the messaging activity of just 16 processors in a very short-running application. AIMS handles longer timelines and more processors by adding scrollbars to the window. Note that such a mechanism does not really assist the user in dealing with largescale data, however, since only a portion of the display can be viewed at a time.

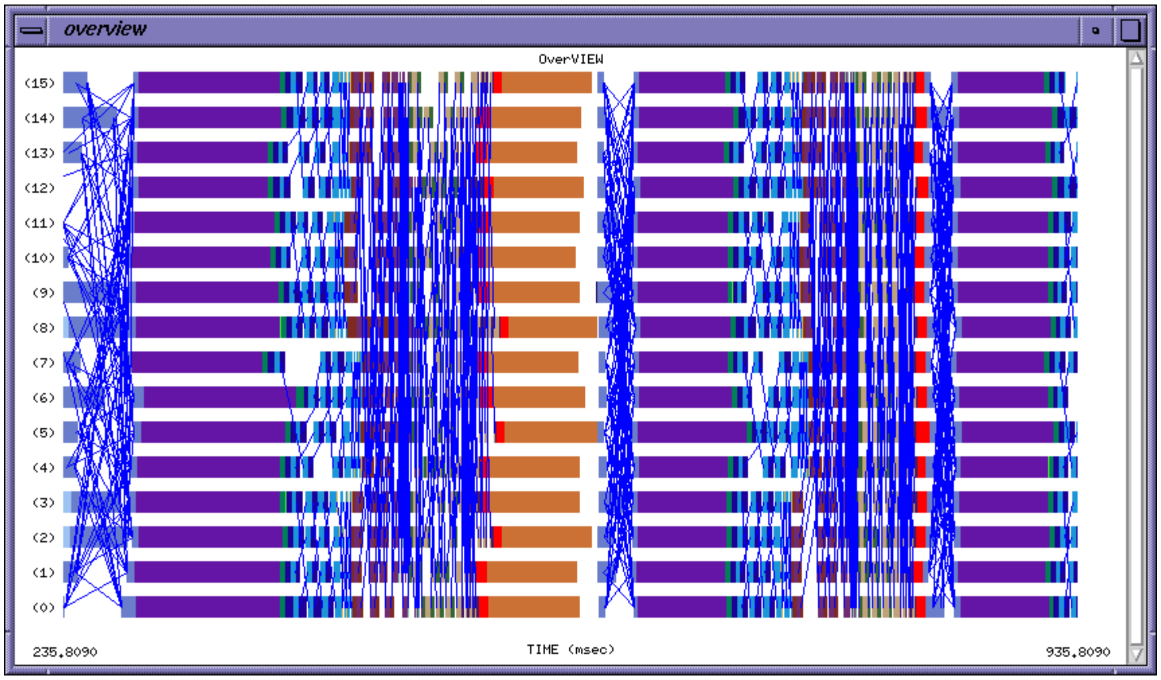


Figure 3. Typical timeline display, showing message exchanges
and computational phases for 16 processors [14].

The display presented in Figure 4 is from xprofiler [3], whose thumbnail sketch assists the user to maintain a sense of context while navigating through portions of a potentially large image. While this is an improvement on simple scrollbars, it is still subject to the basic problem of failing to accommodate complexity. Users must still move through very large volumes of detailed information.

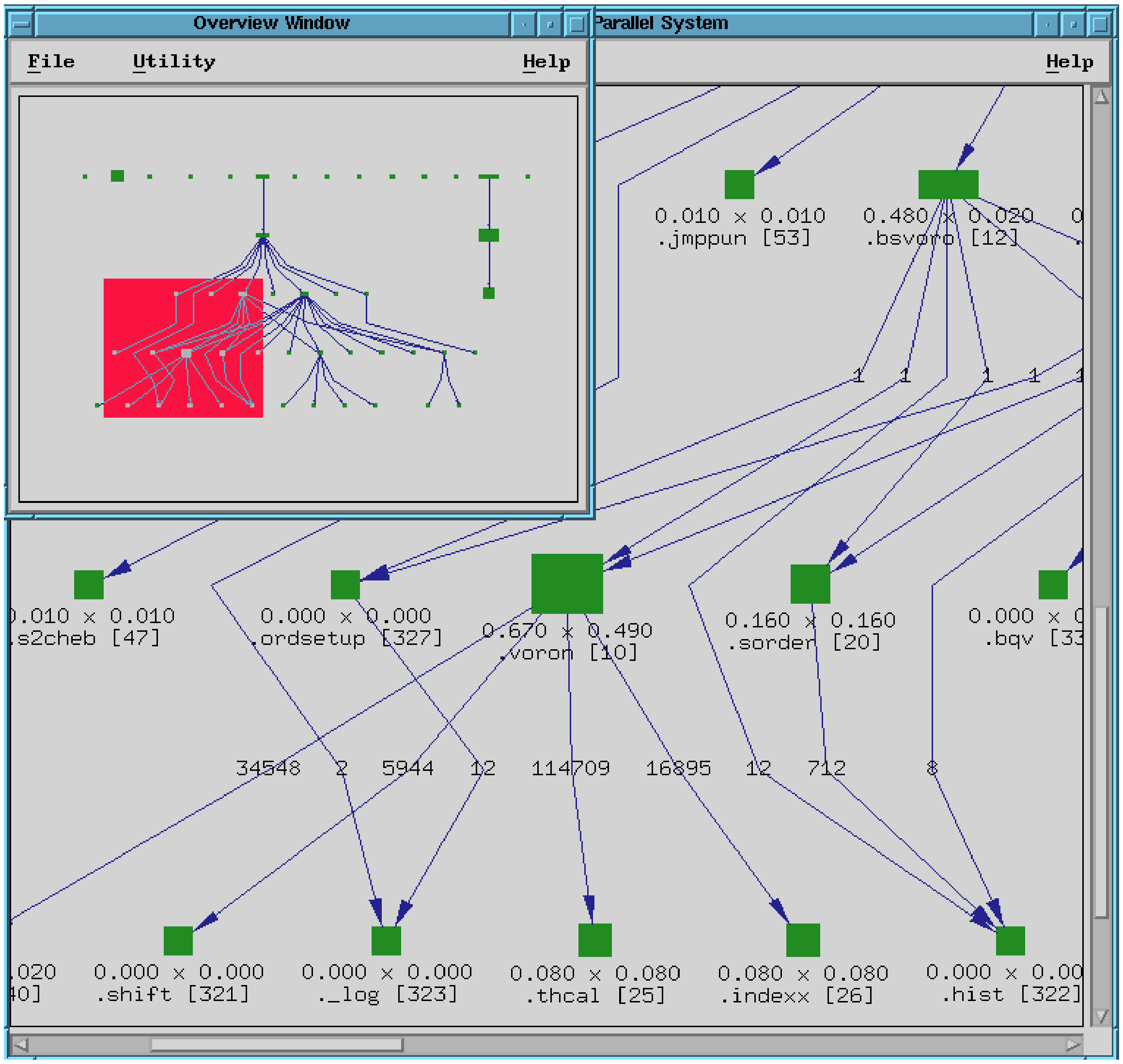


Figure 4. Xprofiler’s [3] thumbnail display assists scrolling,
but the main display is still too cluttered for good comprehension.

A related approach is taken by Jumpshot [15], which augments its timeline displays with a "preview" (see Figure 5) that indicates which portion of the execution trace is currently being displayed. In this case, the display also summarizes some of the data from the timeline (the frequency with which events occurred). Thus, the user can move backward and forward through time to portions of execution that exhibited particularly high or low event frequencies.

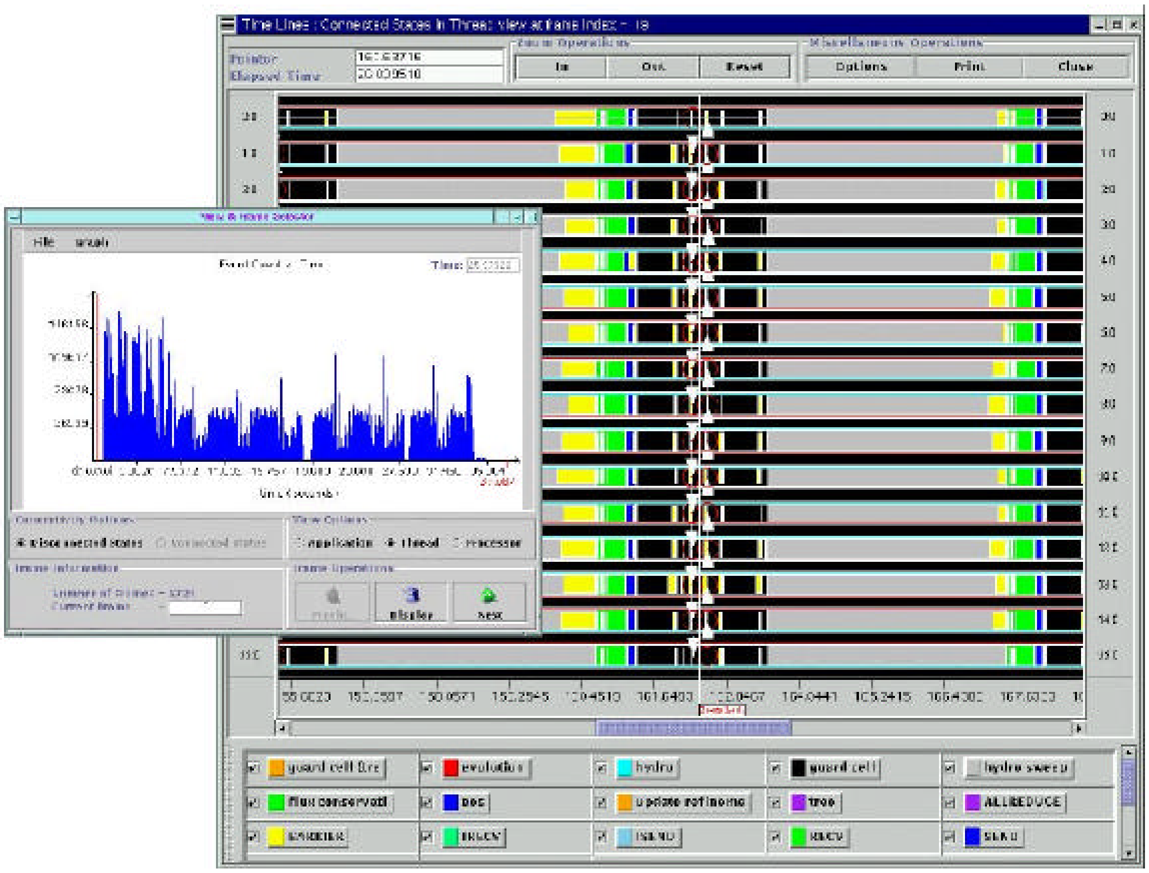


Figure 5. Jumpshot [15] provides a "preview" that facilitates navigation through long timeline displays.

Another problem associated with complex applications is how to correlate program behavior back to the source code where it occurred. Several tools support this by allowing the user to click on an object in one of the graphical displays and view the associated source code. DEEP [1] takes this a step further by providing a "whole program" view, shown in Figure 5. In this pixelized view of source modules, colored lines represent each line of source code, with the color indicating an abstracted performance characteristic (e.g., "Good," "OK?," "Caution," "Bad"). Our own studies of parallel programmers have indicated that this type of representation is highly intuitive; users were able to identify program errors very quickly using similar pixelized displays [7].

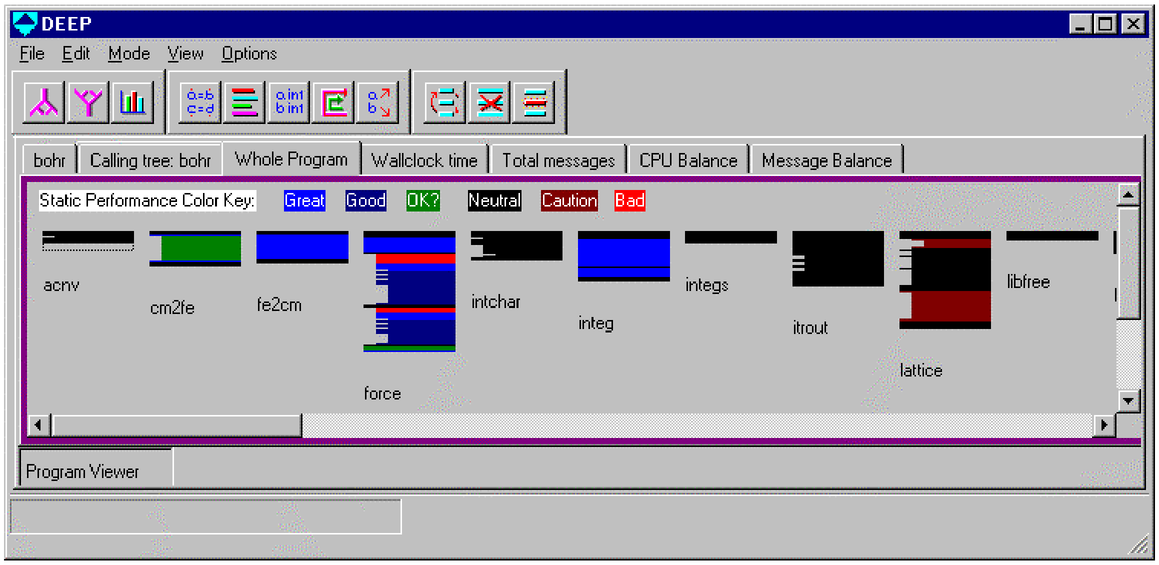


Figure 6. DEEP's [1] source code abstraction offers an intuitive way to compare the behavior of different program units.

In general, performance tools have not been successful in addressing the issue of application complexity. They may make it possible to exclude some of the large quantities of data from a display through basic filtering operations. There are no mechanisms, however, for aggregating groups of related events or measurements into higher-level units. For example, tools with timeline displays (see Figures 3 and 5), should allow the user to combine sequences or nestings of states into coarser aggregates during display. This would relieve the user from having to manually sift through long streams of events and assimilate them to identify meaningful patterns.

Current tools are also lacking in basic search capabilities. It should be possible to search through large displays, for the occurrence of specific clusters of events ("a multicast that went to just processors 0-4"), threshold values ("CPU utilization dropped below 20%"), or compound conditions ("state A occupied more than 10 ms and involved more than 3 processors"). These are the types of activities that users currently perform manually in order to make sense of performance data. To do so using existing tools, the user must make manual notes and/or capture partial screen images for subsequent comparison.

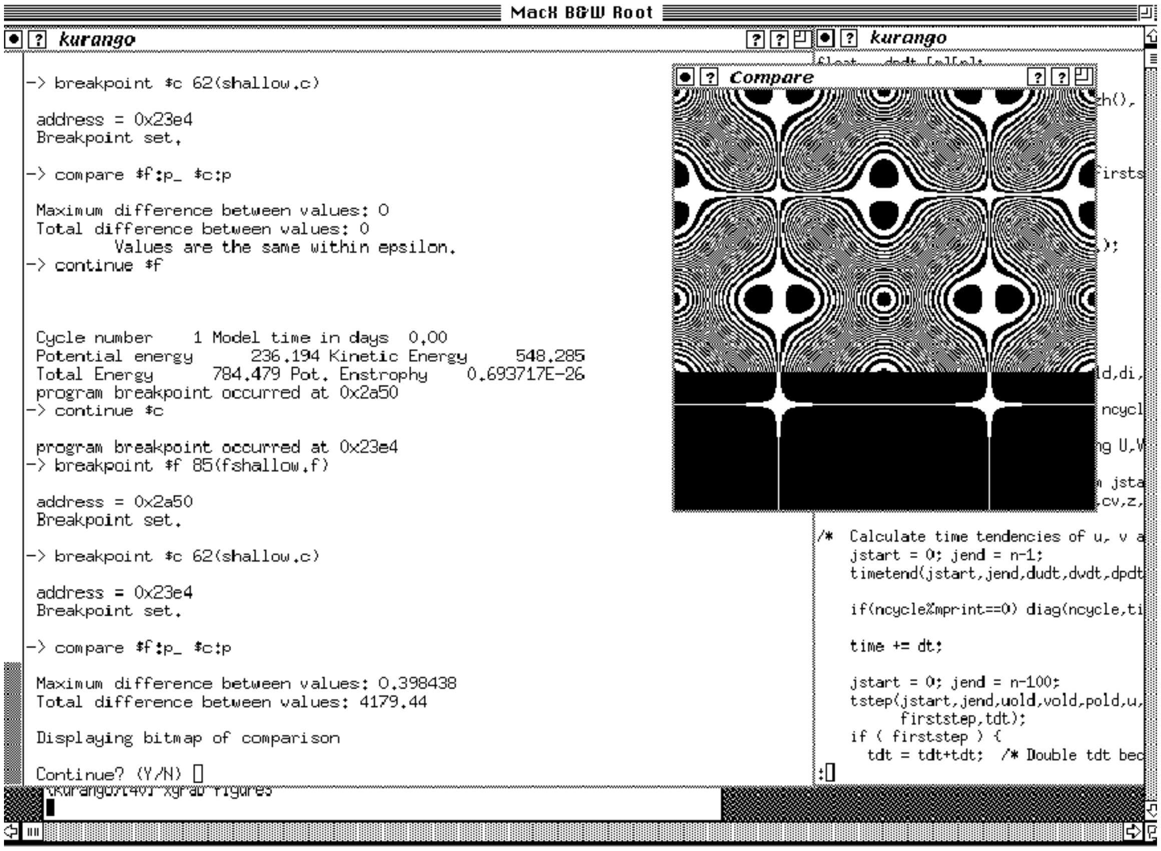
Supporting Suite-Based Performance Analysis

Today's applications also suffer from the characteristic that **portable applications require that testing regimes be much more comprehensive** than those of programs targeted to single platforms. The reasons for this stem from the instability of HPC operating environments. As discussed in the introductory section, even minor changes in a hardware or software platform can cause performance bugs or incompatibilities to surface in an HPC application. For portable applications, those changes increase with each platform supported -- and if the application is heterogeneous, the increase is multiplicative.

The developers of these codes report that they are constantly in a cycle of re-testing and re-evaluating program performance [2, 9]. At each code revision, or each time one of the platforms changes, a new "suite" of program executions is run. The outcome must then be cross-analyzed with comparable suites from other versions and platforms. The process is similar to regression testing, except that it is not just the results of the program runs that must be compared, but also their performance.

Unfortunately, no HPC performance tool is capable of presenting data from more than one program execution at a time. This means that in order to apply tools to suite-based performance analysis, the user must invoke multiple copies of the tool, each applied to a different program run, and attempt to manipulate the copies in synchronized fashion in order to make comparisons.

To meet the needs of today's applications, tools should be able to compare and cross-analyze multiple executions. Visual displays could then indicate where performance differed and by how much, allowing the user to identify the effects of program modifications or platform changes. Consider, for example, the way that the Guard debugger [13] allows users to compare the data values computed during two different executions of the same program. Figure 7 illustrates a simple display of differences; white pixels denote elements whose values differ. A series of display filters are available to accommodate precision differences, allow the specification of boundary thresholds, etc. This type of mechanism would make it possible to ignore some performance differences (say, variations in message length due to a longer, platform-specific message header) while making any other variation in message length obvious.



```
-> breakpoint $c 62(shallow.c)

 address = 0x23e4
 Breakpoint set.

-> compare $f:p_ $c:p

 Maximum difference between values: 0
 Total difference between values: 0
         Values are the same within epsilon.
-> continue $f



 Cycle number    1 Model time in days  0.00
 Potential energy      236.194 Kinetic Energy     548.285
 Total Energy      784.479 Pot. Enstrophy    0.693717E-26
 program breakpoint occurred at 0x2a50
-> continue $c

 program breakpoint occurred at 0x23e4
-> breakpoint $f 85(fshallow.f)

 address = 0x2a50
 Breakpoint set.

-> breakpoint $c 62(shallow.c)

 address = 0x23e4
 Breakpoint set.

-> compare $f:p_ $c:p

 Maximum difference between values: 0.398438
 Total difference between values: 4179.44

 Displaying bitmap of comparison

 Continue? (Y/N) 
```

```
/*  Calculate time tendencies of u, v a
    jstart = 0; jend = n-1;
    timetend(jstart,jend,dudt,dvdt,dpdt

    if(ncycle%mprint==0) diag(ncycle,ti

    time += dt;

    jstart = 0; jend = n-100;
    tstep(jstart,jend,uold,vold,pold,u,
          firststep,tdt);
    if ( firststep ) {
      tdt = tdt+tdt;  /* Double tdt bec
:
```

Figure 7. Tools need mechanisms to compare two sets of performance data, much as the Guard debugger [13] compares array contents from two program executions.

Given the potential complexity of suite-based analysis, it is unlikely that any single performance tool will be flexible enough to meet all needs. To leverage the availability of other data analysis tools, performance tools should be capable of exporting their data in a format that can be input to spreadsheets, databases, statistical packages, etc. Although this is a very simple extension, to date only one research tool has actually implemented it [11].

Across the board, current tools are missing the opportunity to address the needs associated with testing portable codes. Their failure to provide even the most rudimentary support for suite-based performance analysis is a handicap that severely limits tool usefulness.

Conclusions

Current performance tools have made a series of tradeoffs that appear to be based on the assumption that applications are relatively stable and can be tuned on a per-platform basis. This is simply not true of today's HPC applications. Programmers are still concerned with performance, but the ability to run on or across multiple platforms is at least as important. They are unwilling to change their codes to enhance performance when those changes impede portability.

Because of increasing demands for portability, the nature of performance tuning activities is shifting. Suite-based performance analysis, which uses regression testing methods to determine the impact of improvements across collections of platforms, is becoming the norm. At the same time, applications are growing in complexity, quickly exceeding the scaling capabilities of performance tools. Heterogeneous applications, which are also beyond the scope of current tools, are increasing in popularity and likely to continue doing so.

The crux of the problem is that tools are not addressing the real challenges for today's HPC users:

- **Portability at multiple levels:** tools must accommodate a variety of desktop display platforms, as well as applications that execute across multiple target platforms.
- **Performance variability:** tools must be able to indicate where user efforts will yield the most impact.
- **Code complexity:** tools must provide mechanisms for aggregating groups of events or measurements into higher-level units, and for searching through performance data more flexibly.
- **Suite-based performance analysis:** tools must be capable of analyzing -- or at least differencing -- data from more than one program execution at a time.

The gap between tool support and user needs appears to be widening rather than narrowing, forcing HPC programmers to expend time and effort on tasks that could be automated. The challenge for performance tool developers is to focus more attention on the "right" issues -- those that are of most concern to their users.

Acknowledgements

Support for the user studies to identify the requirements associated with code portability was provided by the Programming Environments and Tools (PET) program of the U.S. Department of Defense's HPC Modernization Initiative, through the NAVO Major Shared Resource Center (www.navo.hpc.mil). Additional support was provided by the National Partnership for

Advanced Computational Infrastructure (www.npaci.edu), which is funded by the National Science Foundation.

References

[1] Brode, Brian Q. and Chris R. Warber. "DEEP: A Development Environment for Parallel Programs," *Proc. International Parallel Processing Symposium,* 1998, pp. 588-592.

[2] Frese, Michael H. and Robert E. Peterkin, Jr. "A User/Developer's Perspective on Portability of High Performance Codes," presentation at Parallel Tools Consortium Annual Meeting, April 1999. Available online at http://www.ptools.org.

[3] IBM Corporation, *IBM AIX Parallel Environment: Operation and Use,* IBM Corporation publication SH26-7231, 1996.

[4] Kim, Seon Wook, Insung Park and Rudolf Eigenmann. "A Performance Advisory Tool for Novice Programmers in Parallel Computing," *Proc. Workshop on Languages and Compilers for Parallel Computing,* 2000.

[5] Malony, A. and S. Shendra. "Performance Technology for Complex Parallel and Distributed Systems." In *Distributed and Parallel Systems from Concepts to Applications,* ed. G. Kotsis and P. Kacsuk. Kluwer, Norwell MA, 2000, pp. 37-46.

[6] Pallas GmdH, "VAMPIR -- Visualization and Analysis of MPI Resources," Pallas GmdH publication available online at http://www.pallas.de/pages/vampir.htm, 1998.

[7] Pancake, Cherri M. "Applying Human Factors to the Design of Performance Tools," *Proceedings of Euro-Par 1999,* 1999, pp. 440-457.

[8] Pancake, Cherri M. "Is Parallelism for You? Rules-of-Thumb for Computational Scientists and Engineers," *Computational Science & Engineering,* Vol. 3, No. 2, 1996, pp. 18-37.

[9] Pancake, Cherri M. Unpublished interview notes from field studies conducted with HPC programmers on behalf of the Parallel Tools Consortium, Intel, IBM, Convex Computer Corp., and Dept. of Defense HPC Modernization Initiative, 1995-2000.

[10] Pancake, Cherri M. "Software Support for Parallel Computing: Where Are We Headed?" *Communications of the ACM,* Vol. 34, No. 11, 1991, pp. 52-64.

[11] Park, Insung *et al.*. "Parallel Programming and Performance Evaluation with the URSA Tool Family," *International Journal of Parallel Programming,* Vol. 26, No. 5, 1998, pp. 541-561.

[12] Snelick, Robert. "S-Check: A Tool for Tuning Parallel Programs," *Proc. 11th International Parallel Processing Symposium,* 1997, pp. 107-112.

[13] Sosic, Rok and David Abramson. "Guard: A Relative Debugger," *Software - Practice and Experience*, 1997, 27 (2): 185-206.

[14] Yan, J., S. Sarukhai and P. Mehra. "Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs Using the AIMS Toolkit," *Software - Practice and Experience*, 1995, 25 (4): 429-461.

[15] Zaki, Omer, *et al.*. "Toward Scalable Performance Visualization with Jumpshot," *International Journal of High Performance Computing Applications,* Vol. 13, No. 3, 1999, pp. 277-288.