NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making
of photocopies or other reproductions of copyrighted material. Any copying of this
document without permission of its author may be prohibited by law.

Visualizing Performance Debugging

Ted Lehr*, Zary Segall, Dalibor Vrsalovic, Eddie Caplan, Alan L. Chung, Charles E. Fineman

April 1989
CMU-CS-89-140

School of Computer Science
and *Department of Electrical and Computer Engineering
Carnegie Mellon University

Abstract

This article discusses visualizing performance of parallel and sequential computations using the Programming and Instrumentation Environment (PIE) as an example. First, the importance of being able to visualize the performance of a computation is demonstrated. Secondly, an easy guide to one way, the PIE way, of his visualization process is presented. This is followed by examples of concrete uses of the environment. Finally, some of the pressing issues concerning measuring performance are discussed.

Three issues are stressed in this article:

1. Visualization of program performance is necessary for fast, correct performance debugging.
2. PIE is a powerful and useful implementation for supporting visualization. It is also a dramatic teaching assistant in that it visually shows actions like forking, scheduling, etc.
3. PIE provides complete support for visualization. The system is aware of issues like program replay, suitably presenting performance data, compensation for perturbation by the measurement facility and support for multiple architectures and languages .

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), Arpa Order No. 4864, monitored by the Space and Naval Warfare Systems Command under contract N00039-87-C-0251, and in part by the National Science Foundation, Grant No. CCR-86-02-143.

The views and conclusions contained in this document are those of the authors and and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation or the U.S. Government.

Errata:

On page 7 of the text:

- The textual references to "yellow rectangles" are shown in Figure 2-4 as dotted rectangles.
- The textual references to "green rectangles" are shown in Figure 2-4 as slashed rectangles.

1. Introduction: Visualizing Performance

Designing computations to perform efficiently is usually an iterative task in which programmers alternate between measuring and modifying the performance of successive computation prototypes. This iterative performance tuning is called *performance debugging* and can be done in several ways. Perhaps at the start, programmers may gather general *statistics* on metrics such as the average amount of parallelism in their computations. Averages are large grain reports of what computations do and, as such, often reveal only the hints of trouble. An average parallelism of, say 2.1, in a computation intended to have eight processes running simultaneously probably suggests a problem. The average, however, does not point directly to where the problem lies. It can be misleading for it may have remarkably little to do with "reality." There is no guarantee that a value was measured that equals the average and it may not even be possible for such a value to ever occur. Embellishing averages with other statistics such as measures of spread is an improvement but may raise more questions than they answer. The additional statistics may suggest that the degree of parallelism exhibited by a computation is not only small but also varies widely, raising the specter of erratic performance. Does the computation occasionally hit synchronization barriers, forcing some of its processes to wait? When does it do this and how often? These are questions statistical information can raise, but is hard pressed to answer.

Because of the limited information contained in averages and other statistics, programmers must pry into the collection of data that yields the statistics. They must sift through the records, searching for suspicious data that might indicate anomalous instances of exceptional or degenerate performance. They must look at individual cases of scheduling and communication costs. If they have not collected the proper data, additional experiments must be run. All in all, a time consuming and tedious endeavor. The tedium arises because of the human limitations in conceptualizing relationships between mountains of raw numerical data and the objects they characterize; in this case, computational constructs. It is essential that this tedium be eliminated if performance debugging is to be a productive activity. The raw numerical data must be distilled into a form that is readily grasped by programmers. In order to productively debug the performance of their computations, programmers must be able to *visualize* that performance.

Just as measuring performance can be done in a number of ways, visualization of performance takes on several forms. For example, the histogram in Figure 1-1 shows the distribution of the degrees of parallelism in our hypothetical eight-process computation. It is more revealing than straight numerical statistics in alluding to the possibility of a performance problem. It is fairly clear, for example, that the computation never even comes close to eight-way parallelism. The histogram does not tell, however, when the various degrees of parallelism occur or what the rest of the computation is doing at those times. Trying to answer the question of "when", the performance view shown in Figure 1-2 further sharpens the debugging scalpel by plotting the amount of parallelism versus time during an important period in the computation. The edge which this kind visualization has over the histogram is clear. A programmer *immediately* sees how much parallelism the computation is able to muster at different moments in its execution. This is the primary advantage of presenting performance data in a visual form: programmers interpret performance data more quickly. It is quite obvious, for example, that at the start of the measured period the computation manages to get parallelism of about four or five but quickly plummets to a fairly steady level of around two. Even though this time-line helps to answer the question of when parallelism occurs, a problem remains: the programmer needs more information in order to determinine precisely where this behavior occurs.

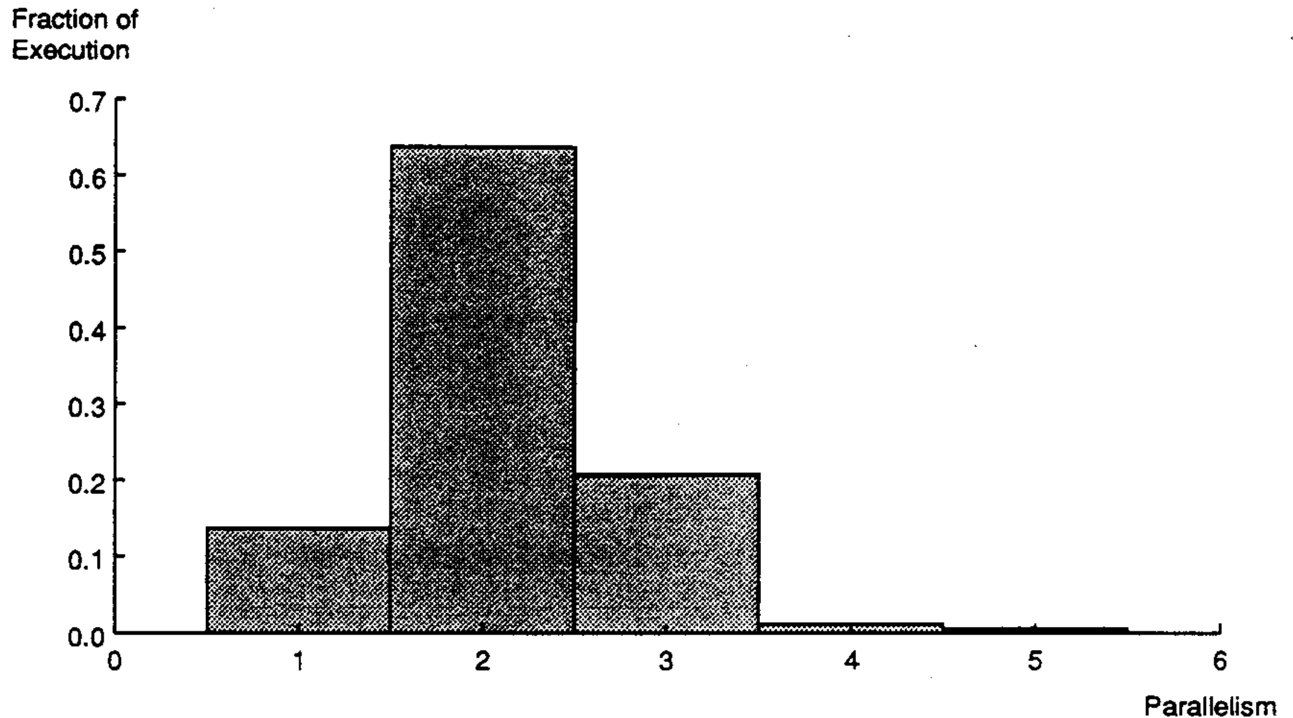


Figure 1-1: Distribution of Parallelism in Computation (measured at regular intervals)

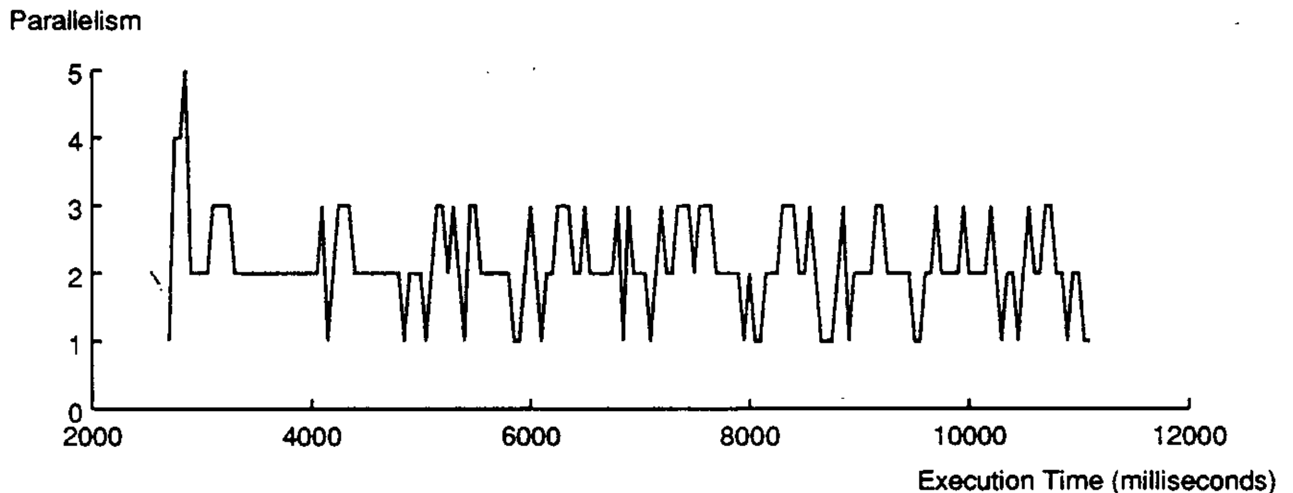


Figure 1-2: Parallelism versus Time

Statistical plots, such as histograms, help to guide programmers to the presence of lackadaisical performance by revealing subtle hints of the underlying problems. Histograms, however, cannot lead programmers directly to where the performance difficulties lie. Plotting the data on a time-line helps to focus the suspicions generated by statistical data, but several must be studied together so that programmers see a more global picture of what is going on. If one time-line reveals an anomaly, several may help programmers isolate not only what happens when the anomalies arise, but also the succession of events that leads up to them. In order for programmers to debug for performance they need to be able to visualize executions on the level they program in. Because programmers know their programs by the constructs and abstractions with which they designed them, a vital characteristic of the presentation of performance data must be a visual mapping which vividly draws the connection between the data and the computational constructs which are responsible for them.

`Raw data >> graphics >> mapping magic << visual icons << program text`

As is diagramed here, if programmers are to visualize the performance of computations, not only must the performance *data* have visual representations, but so must each *construct* in the corresponding programs. The remainder of this paper focuses our arguments for performance visualization by examining a special software development environment, called the Parallel Programming and Instrumentation Environment (PIE). PIE is meant for developing performance efficient parallel and sequential computations[1]. Following an explanation of the general features of PIE, we reveal that the hypothetical eight process computation we have been discussing is actually a real one and we use PIE's visualization tools to isolate and repair its parallelism problem. After showing how valuable PIE is in fixing the computation's parallelism problem, we discuss two other, more difficult examples of using PIE. The two examples use slightly modified versions of the repaired computation executing under different circumstances. The first focuses its examination on the behavior of a uniprocessor's kernel running under two different scheduling algorithms. The second is an instructive glimpse of how two simultaneously executing computations behave on a multiprocessor after they spawn more parallel processes than there are processors to run them. We then round out our discussion of performance visualization by remarking on some of the issues involved in correctly presenting visual information.

[1] A performance efficient computation is a computation which decomposes onto a target set of computing resources in such a way that that the resources are matched to the functions of the computation.

2. A Visual Programming Environment: The PIE example

Automatic assistance for visually projecting performance data onto programming constructs is practically necessary if visualization is to quicken the performance debugging task. Bringing together programs and their performance data is best done by systematic software development environments which integrate different data analysis tools into one package providing a "computational laboratory" in which programmers can easily design experiments to test the behavior of their computations. A software development environment for performance debugging is not an effective one if it compels its users to vigorously *search* for problems, especially for problems that could be easily revealed by automatic techniques. On-the-other-hand, an environment should not attempt to second guess a designer by reporting that a certain set of performance data represent a problem. An environment which impudently makes qualitative judgements about a computation's performance without the user's permission may annoy and mislead the user. The proper role of an environment, then, is to present the information it retrieves about computations in forms that assists users in making their own qualitative judgements about how their computations behave. The Parallel Programming and Instrumentation Environment (PIE) [16] is a software development environment for performance debugging which gives programmers ways to observe how computations execute by making use of special development and run-time visualization tools. PIE is not just a programming environment, as defined by Dart and others [6], which supports only the development of program *coding*. PIE supports a software development methodology extended to the analysis, verification and validation of a computations performance.

Where does the environment begin to assist a programmer? Well, let us assume the programmer has a problematic parallel matrix multiplier computation executing on a 16-processor shared-bus architecture that exhibits a parallelism problem like that described earlier. The programmer wants some performance information on it; specifically, why is the computation's average parallelism only 2.1 especially since eight multiplier sub-processes are spawned, each of which operate on well partitioned parts of the matrices. Each of the computation's sub-processes examines the size of the parts of the two matrices it is passed and decides whether the parts are small enough for it to operate on without partitioning them further and passing them on to another sub-process which it spawns. Upon making the decision, the sub-process iterates through its part of the matrices, multiplying each pair of row and column and writing its result out to shared memory.

Figures 2-1 and 2-2 showcase how PIE implements visualization of programming constructs. Figure 2-1 depicts parts of the text of the computation via three windows of a special PIE editor. The program is written in an extended C-language, called MPC [19], which supports parallelism using constructs that implement actions such as sharing of global memory and spawning of parallel processes. It is not important to fully understand the semantics of the text but some elucidation of the unconventional parts will be helpful.

Briefly, the top window shows a section of the definition of the computation's multiplier procedure, `multproc`. It includes a variable declaration of the type `multiply` which is an instance of what MPC calls an activity or `act` as shown in the middle window. Activities can be thought of as process-like units of parallel work which, when spawned from the same program, are able to enjoy such luxuries as sharing and operating on global memory. Notice that `multiply` contains a call to `multproc`. `Multproc` implements the basic matrix partitioning and multiplying functions described above. After the value of an element of the result matrix is calculated, it is written out using `put`, shown in the lowest window, which is an instance of an MPC function type called `opr`. Entities of this type may be shared by several activities

```
multproc(x1, x2, y1, y2, mx, my, sz)
     int              x1, x2, y1, y2, mx, my, sz;
{
  int               ex, ey, i, j, k;
  float             t, tmp, tmp2;
  multiply          subtask;

  ex = x2 - x1 + 1;
  ey = y2 - y1 + 1;
  if (ex > ey) {
    if (ex > mx) {
      subtask (x1, (x1 + ex / 2 - 1), y1, y2, mx, my, sz);
      multproc((x1 + ex / 2), x2, y1, y2, mx, my, sz);
      join(subtask);
      return;
    }
    if (ey > my) {
      subtask (x1, x2, y1, (y1 + ey / 2 - 1), mx, my, sz);
      multproc (x1, x2, (y1 + ey / 2), y2, mx, my, sz);
      join(subtask);
      return;
    }
--**-PIEmacs: matsync.mpc          (MPC)--34%--{ Normal }--------------------
act
multiply(x1, x2, y1, y2, mx, my, sz)
     int         x1, x2, y1, y2, mx, my, sz;
{
  multproc(x1,x2,y1,y2,mx,my,sz);
};
--**-PIEmacs: matsync.mpc          (MPC)--31%--{ Normal }--------------------
  opr float          put(x, y)
    int               x, y;
  {
    sync(put){
        export(matrix_data[x][y]);
      }
  }
--**-PIEmacs: matsync.mpc          (MPC)--23%--{ Normal }--------------------
```

Figure 2-1: Part of a Matrix Multiply Program Text

and are used to operate on global memory. The only feature of `put` that needs to be appreciated here is the `sync` function, which is a special MPC function that enforces mutual exclusion on global operations. Here, `sync` ensures that only one result may be written back to global memory at a time.

Figure 2-2 is another PIE window showing the visualization of the matrix multiplier's principal constructs. Each box in Figure 2-2 has a corresponding textual entry shown in Figure 2-1. In fact, when a box is touch-selected by a mouse, as is shown by the enlarged border surrounding the box labeled `[c] multproc`, the editor window automatically moves its cursor to the head of the corresponding textual construct, in this case, a call to the `multproc` procedure.

Having had the program's constructs automatically mapped onto a visual representation, it is time for the programmer to gather performance information. Let us assume that the programmer has run some tests, either using PIE or other methods, and already knows that the parallelism in the computation is disappointing. PIE can generate performance views like the histogram and timeplot shown in Figures 1-1 and 1-2, but these are ancillary to a grander, more informative format which will be shown shortly. Since the programmer wants to improve the parallelism of the computation, it is important to look at what each multiplier process does when it executes. The programmer knows that if the computation has eight

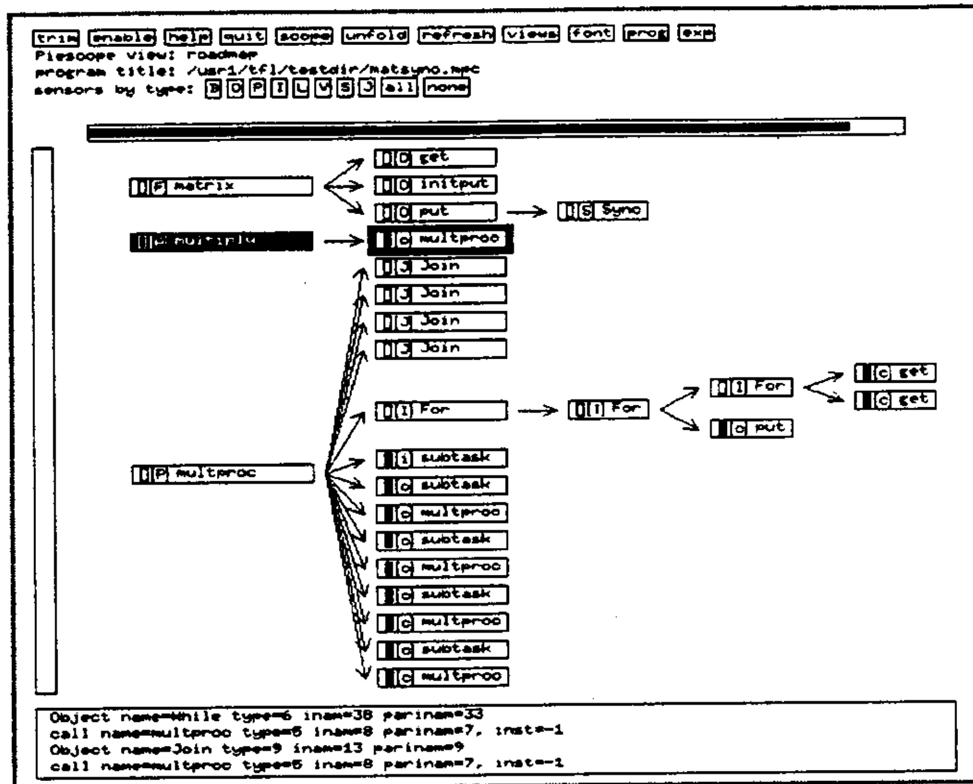


Figure 2-2: Part of Visual Representation of Program

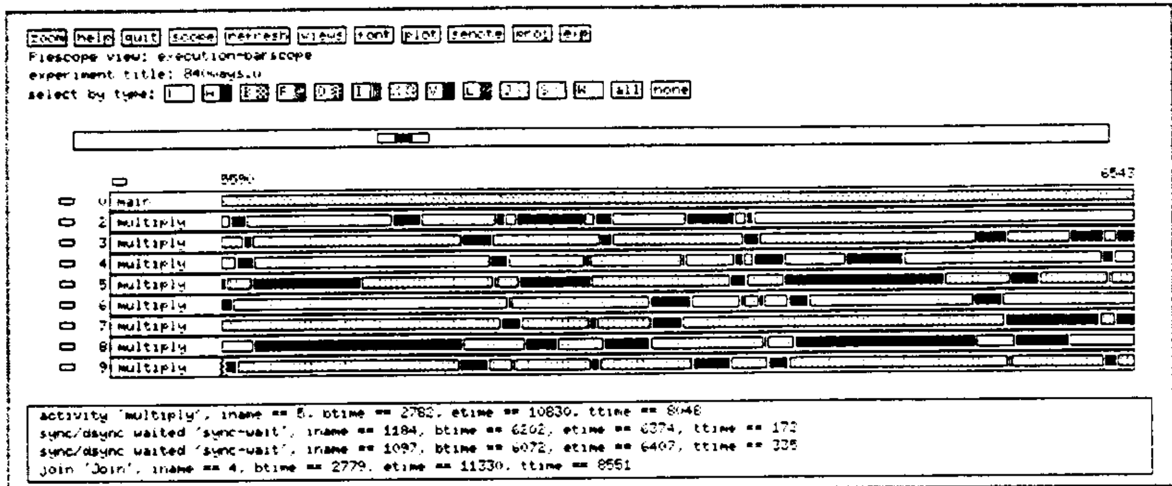
processes eligible to run in parallel but only two or three ever seem to be able to do so, then the others must be blocked and waiting for some event to occur. The programmer thus decides to examine the behavior of any program construct that might force a multiplier to wait, namely the `sync` just discussed and the `join` (an example is shown in the top window in Figure 2-2) which a multiplier executes after finishing its part of the matrices and moves to join its children. To get this information with PIE is simple. Figure 2-3 shows a number of darkened boxes, `[A] multiply`, `[S] Sync` and several cases of `[J] Join`. The `[A] multiply` represents the multiplier processes and `[S] Sync` is the synchronization function in the `put` operation discussed earlier. Each `[J] Join` represents in instance of a `join` function. The darkening of these boxes indicates that the programmer, using a mouse, has selected them to be automatically observed when the computation runs.

We have only completed half of the mapping of the program constructs onto performance data. Before it is completed, of course, the programmer must run the program to get some data. PIE ensures that when a selected construct executes, information is collected which identifies the construct, the process in which it executed and how long it executed. For example, time stamps are retrieved at the beginning and ending of each multiplier process (`multiply`). The time stamps are then married to other information uniquely identifying each `multiply` instance. As noted earlier, PIE presents the performance information in a variety of ways including histogram and time-line formats. Although formats like those shown 1-1 and 1-2 are incomplete and imprecise representations of the behavior of computations, they allow the wealth of performance data to be distilled into visual forms for quick and tractable detection of performance problems. More precise identification of what those performance problems are is better done using the upper view shown in Figure 2-4, PIE's principal format for visualizing performance data.

Figure 2-3: Using the Visual Representation to Enable Sensors

In this view, time is measured in milliseconds on the horizontal while the processes of the computation are ordered on the vertical. This particular view shows only the part of the execution from 5.590 to 6.543 seconds. The execution of each process is depicted by a concatenation and occasional overlap of several rectangular patterns, each representing a particular episode in the process's history. Waits due to a `sync`, for example, show up as yellow rectangles lying on top a dark, extended rectangle representing the generic body of the associated process. Green rectangles represent periods when processes are waiting to join a child. If one entity is contained within another, its rectangle overlaps the rectangle of the other. For example, since `sync`-waits execute within processes, all yellow `sync`-wait rectangles overlap their corresponding process rectangles. When any of the rectangles is selected by the mouse, the cursor in the PIE editor is automatically moved to the head of corresponding construct in the program text. Here for example, the `join` rectangle in main has been clicked on (as shown by the "Join" message in the bottom of the small text window immediately beneath the rectangular patterns). The cursor in the corresponding PIE editor window, shown in the bottom view of Figure 2-4, is moved to the head of the appropriate `join` definition. The programmer can now analyze the performance based on data automatically projected onto the computation's structures. The mapping of performance data onto program constructs is complete!

What does the upper view in Figure 2-4 depict? It represents a small section of the computation's execution when only some of all the row/column multiplications performed by the entire computation are executing. The initial process of the computation, `main`, shown in green here, is waiting to join the other processes. Before entering this stage of the computation, it performs the I/O for the computation as well




```
main(argc, argv)
     int             argc;
     char            **argv;
{
  int             sz, mx, my;
  multiply        task;

  if (argc != 4) {
    fprintf(stderr, "Usage: matrix size dx dy\n");
    exit();
  }
  sz = atoi(argv[1]);
  mx = atoi(argv[2]);
  my = atoi(argv[3]);
  init_matrices(sz);
  SENSOR("Before task");
  task(0, sz - 1, 0, sz - 1, mx, my, sz);
  join(task);
  print_result(sz);
}
-----PIEmacs: matsync.mpc          (MPC)--Bot--{ Normal }-------------------
```

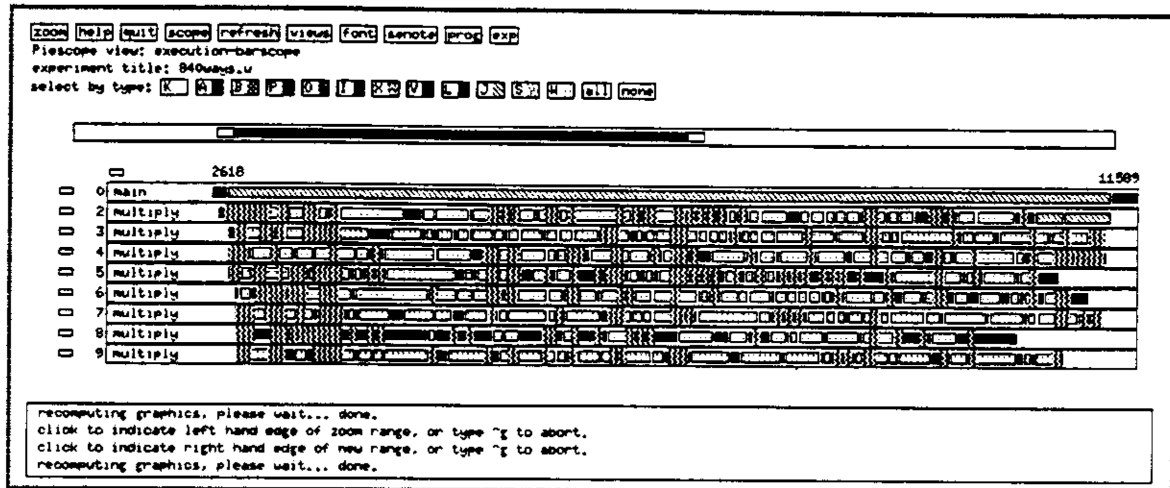
Figure 2-4:

Top: Part of a Parallel Execution of the Computation on 16-processor Machine
Bottom: A PIE editor window

as forks off the first `multiply` process. The `multiply` processes, numbered in the order they are forked off, execute the `multproc` procedure. A special process, one used in gathering performance data on the computation, has been left out of the view in order to simplify this example (it would have been numbered as "1"). Just as expected, the figure shows that, at any given time, several of the processes are waiting. Cutting a vertical swath through a view at any point, for example, slices through only two or three black rectangles indicating that only those processes are doing useful work. Obviously, the source of the performance degradation is the `sync` function in the `put` operation.

What the programmer does to remedy this is up to him or her. Is the `sync` really necessary? Each time the `put` operation is called, after all, it only fills a single location in the result matrix. Since the matrices are well partitioned, no other call to `put` will touch that location. Consequently, the `sync` is not necessary. Figure 2-5 shows the executions of the original and a corrected version of the computation. The top view shows the entire multiplying section of the original computation. The bottom view shows the

corresponding section in the corrected computation without the `sync`. The "squiggles" in the top view represent moments of insufficient resolution on the part of the display, but are otherwise unimportant. Without the `sync`, not only is the average parallelism of the computation greater, but the execution time has been cut from about nine seconds to just under one second. Figure 2-6 is a PIE generated parallelism plot for the corrected computation.



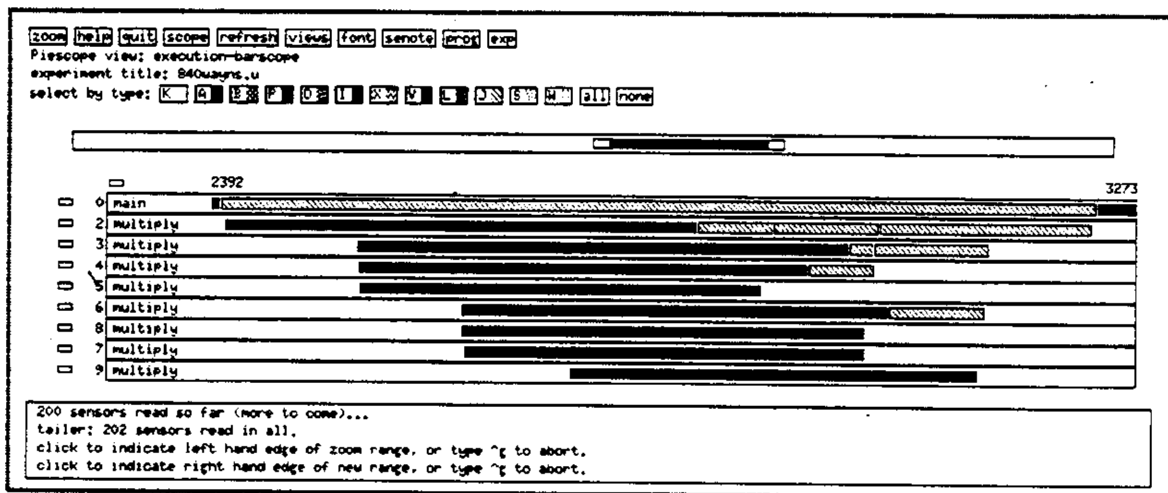


Figure 2-5: Parallel Executions of the Computation with and without the sync

This example was simple and perhaps contrived; the programmer's error of enforcing mutual exclusion on the matrix writes is probably a mistake only a novice would make. Never-the-less, it has shown the value of being able to visualize what a computation is doing. Our next two examples are more rigorous and detailed illustrations of the value PIE brings to both designing and understanding computations. Unlike our introduction, they are not examples of using PIE to find performance bugs in computations. Instead, the thrust of their attention is devoted to examining and explaining the behavior of kernels on a uniprocessor and a shared-bus multiprocessor. The first discusses scheduling performed by two versions of the same operating system running on identical machines. The second describes one instance of what happens when two parallel computations running on a multiprocessor spawn more processes than there are processors on the machine. Together they are dramatic demonstrations of the luxurious advantage of being able to visualize the performance of computations.

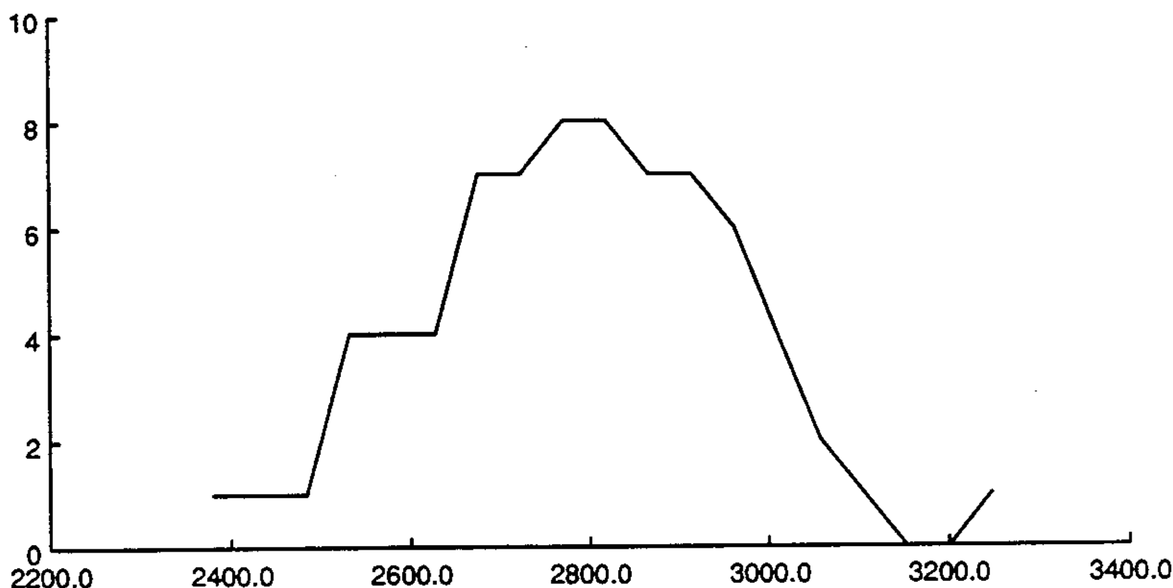


Figure 2-6: Parallelism versus Time for Corrected Computation without sync

3. Visualization Examples: Its Analytical and Pedagogical Uses

The simple example just discussed is not a particularly exciting instance of using the PIE environment. The next examples focus on kernel activity because kernel behavior has accounted for some of the most graphic illustrations of how a significant component of either a program, kernel or system state can affect a computation's behavior. In the first example, the component is a modified kernel scheduler. In the second, it is a system state marked by too many processes. These examples should be of interest to a wide audience because kernel performance impacts the performance of every computation.

The first example analyzes what effect a change in a kernel's scheduling algorithm has upon a computation. The computation is a matrix multiplier similar to the corrected version just discussed. The kernel is Mach [1], an operating system under development at Carnegie Mellon University for supporting parallelism. The computation and kernel both execute on a micro-Vax II is booted in single user mode in order to reduce the number of processes that compete with the computation for the machine's processor.

The second example is most difficult one we present. It is a dramatic illustration of what happens when two parallel computations compete for the same processors on a shared-bus multiprocessor machine. The selected PIEscope views of the example are pedagogic glimpses of what a multiprocessor scheduler does when the number of schedulable, parallel processes[2] is greater than the number of available processors.

In order for the discussion of the examples to be clear, however, it may be prudent to peruse some summary remarks about Mach and its unique context-switch monitor as well as a brief tutorial on basic scheduling issues. Of course, if these topics are already familiar, the remarks can be skipped in favor of studying the two examples.

[2]Shortly, we will define another term, "thread," to replace "process" as the term designating a schedulable entity.

3.1. Mach: An operating system for distributed computing environments

The Mach kernel integrates support for networks of uniprocessors and multiprocessors while presenting a Unix style software environment. The basic Mach primitives support Unix functionality by placing it *outside* the Mach kernel. Although Mach has a number of abstractions for supporting networks and multiprocessors, only the *task, thread, port* and *monitor* abstractions need to be discussed here in order to more clearly understand the upcoming examples.

3.1.1. Tasks, Threads, Ports and Monitors

A task is an address space and a set of system resources (eg. file descriptors) while threads are the basic units of execution. Each thread can be thought of as a program counter and register set. Each task has at least one thread associated with it (a Unix process can be emulated as a single thread executing within a task), although a task may have several threads may threads sharing its resources in parallel. A *port* is the Mach communication abstraction for for sending and receiving typed messages between Mach entities such as tasks. A port is basically a protected queue with associated *send* (enqueue) and *receive* (dequeue) rights. The port abstraction supports intertask communication across network boundaries It is by means of the task, thread and port abstractions that Mach implements much of its support for parallel computing. Specially equipped versions of Mach permit the creation of kernel *monitors* for detecting and recording the context-switches of selected threads [11]. By "monitor," we mean a data collection facility for retrieving run-time information about the execution of computations. These monitors perform observational functions rather the mutual exclusion errands typically ascribed to objects referred to as "monitors."

3.1.2. Other Abstractions

In addition to the task, thread and monitor abstractions, Mach has other primitives for supporting networks of uniprocessors and multiprocessors including portable virtual memory support. For information concerning Mach, see [7], [10], [15], [21].

3.2. Schedulers

Schedulers are operating system functions which assign processes (threads) to processors. They make their assignments using algorithms rooted in polices like first-in-first-out, round-robin, shortest-job-first and others [8], [18]. The policy which a scheduler uses depends upon which computing goals wish to be supported. A scheduling algorithm that helps a single computation perform superbly often has little in common with one that attempts to keep every processor busy in a multi-programming environment. A common desire of all schedulers, however, is reducing the number of context switches. As will be seen shortly, meeting every scheduling goal is not always easy.

3.3. Examples of Kernel Visualization: Understanding the Figures

Visualization, especially visualization of kernel behavior, is useful for comparing performance of architectures or operating systems. Different versions of the same kernel using different scheduling algorithms, for example, frequently yield markedly dissimilar visual views.

Figures 3-1 through 3-5 are selected PIEscope views of a matrix multiplier computation executing on three similar, but slightly different versions of Mach. The PIEscope views show a layer of performance information, the kernel layer, not included in the introductory example. It also shows an additional thread,

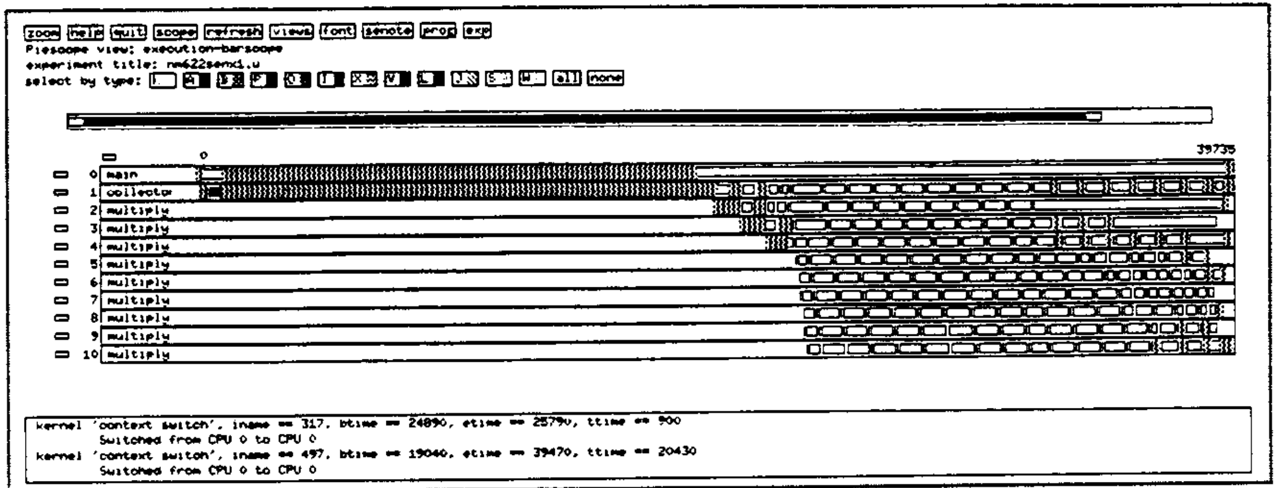


Figure 3-1: Old Kernel: Entire view of matrix multiplication

the `collector`, which, as part of PIE, records user and kernel events belonging to the computation. Although spawned by `main`, its creation and termination are accomplished independently by special libraries linked with the computation at compile time. The `collector` was not shown or mentioned in the earlier example because it would only have cluttered the presentation.

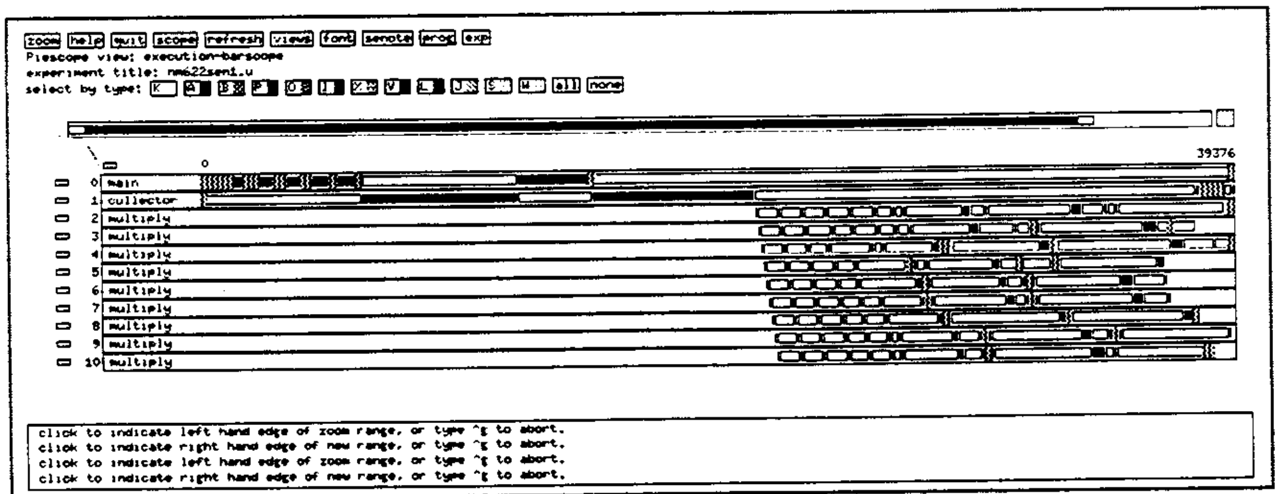


Figure 3-2: New Kernel: Entire view of matrix multiplication

The addition of kernel events to the views needs to be explained. Shaded rectangles represent periods when threads are running. White rectangles represent the periods when threads are switched out. Because the figures depict uniprocessor executions, cutting a vertical swath through a view at any point would slice through only one running thread ... only a single dark rectangle. In parts of some of the views, in Figure 3-1 for example, there are slightly confusing sets of consecutive white rectangles. Between these apparently contiguous periods are comparatively shorter episodes when the associated thread is

actually running but the view does not have the resolution to show this[3].

3.3.1. Time Sharing on three Mach kernels

Figure 3-1 depicts micro-Vax II execution of the matrix multipler on a kernel officially labeled as **XF29** but which we refer to as **Old**. Figure 3-2 depicts the the same computation on on a newer kernel designated as **CS3c**, hereafter referred to as **New**. Figures 3-3 and 3-4 are "zoom" views showing greater detail of each computation. Figure 3-5 is a zoom view of the computation executing on a kernel very similar to New but with an improved thread priority evaluation policy. Each higher resolution view also contains a pair of "metering" lines which measure the time between them. Although the two kernels are dissimilar in a number of respects, our comparison concentrates only on the difference between their scheduling policies.

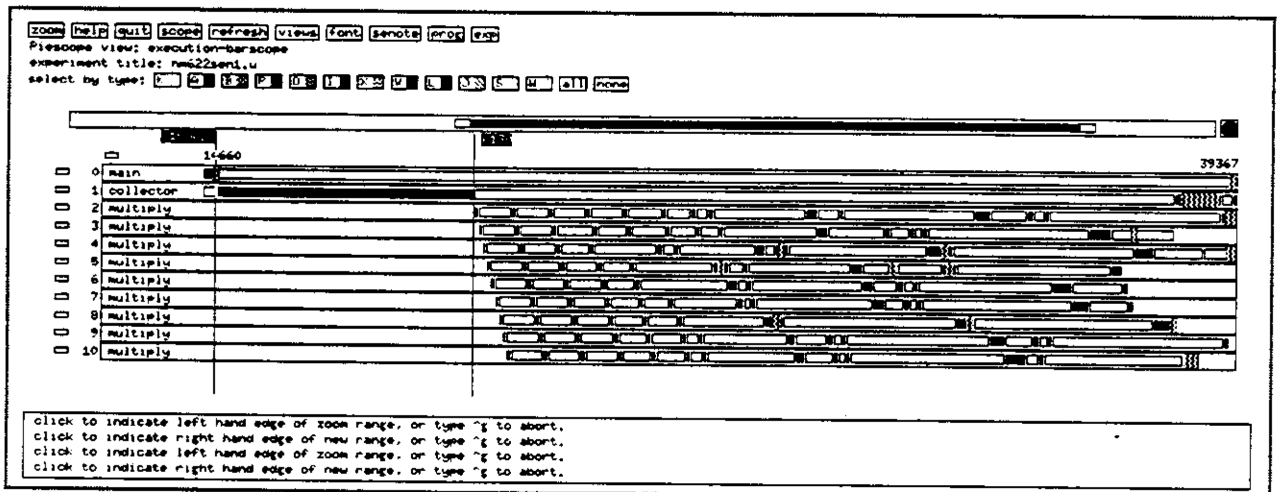


Figure 3-3: New Kernel: A Zoom view of matrix multiplication

As can be seen in Figures 3-1 and 3-2, the computation does not behave identically on the two kernels. Figure 3-1 shows that Old's scheduler generally gives short, uniform time slices to its threads (here `main` is waiting for its children to terminate and is not rescheduled until that time). Old uses a simple scheduling algorithm which usually switches threads every 100 milliseconds. On-the-other-hand, Figure 3-2 shows that New occasionally gives scheduled threads execution time slices of several seconds as evidenced by the initial time slices allocated to `main` and the `collector`. New's scheduler uses a progressive algorithm which gradually increases the length of the execution time slices it allocates causing fewer context-switches per unit time than the old scheduler. This experimental algorithm tracks the history and load on the machine for deciding whether execution time slices can be extended. Figure 3-3 vividly shows this allocation of gradually longer time-slices. During the first moments after the `multiply` threads are spawned, there is regular and frequent context-switching among all the threads, much like the behavior of the threads on Old. After some time, the new kernel's scheduler gradually schedules each thread for longer and longer periods before switching them out. Because there is no significant competition from other threads during the computation's execution, the scheduler determines that each thread may run for longer periods without being unfair to any of them.

---

[3] "Squiggles" are not used in these cases because the number of events beneath the resolution is small.

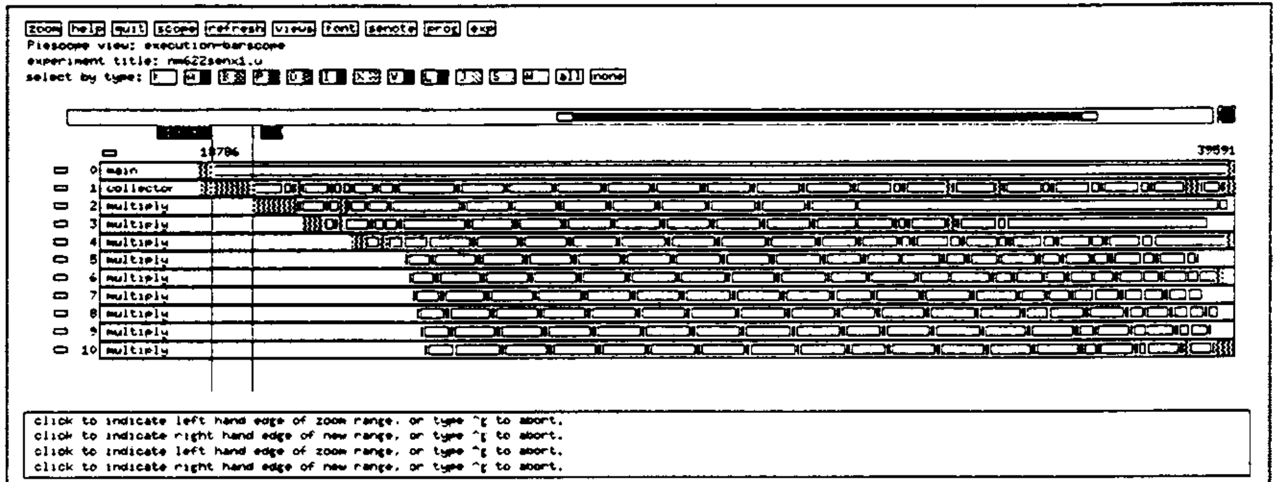


Figure 3-4: Old Kernel: A Zoom view of matrix multiplication

In addition to the advantages New enjoys over Old, there are some drawbacks as well. Threads running on the new kernel seem to suffer from context-switch "flutter" where a thread frequently context-switch to itself as shown in Figure 3-2. The most pronounced episode occurs in the first time slice allocated to `main` and in several of the smaller time slices allocated to the `multiply` threads as evidenced by the "squiggles" in those slices. Such behavior has been seen in other views of executions on New. The New kernel also suffers some problems in equitably time sharing threads. Examining Figure 3-3, it can be seen that just before the first `multiply` thread is forked off, New switches out the `collector` and does not run it again until nearly all the other children terminate. Alternately, Figure 3-4 shows that Old time shares the `collector` with the freshly spawned children.

3.3.2. Comparing Forking Behavior On Old and New

How threads are scheduled after they are running is not the only difference between the old and new kernels. There is also a difference in the forking performance of the schedulers. Figures 3-3 and 3-4 show quite clearly that New successively spawns the first three `multiply` threads more quickly than Old. They also show that the new kernel eliminates the context-switch "flutter" that occasionally occurs in Old just as a thread is forked off. Figure 3-4, for example, shows `multiply` threads 2, 3, and 4 undergoing considerable context-switching soon after they are spawned by Old. The `collector` undergoes similar behavior. Interestingly, much of the context-switching is of the thread to itself, much like the behavior described earlier for the `collector` on New. There is a difference, however. In the case of Old, threads usually context-switch to themselves only for a short time after having been spawned. Threads running on the new kernel, however, context-switch to themselves in an unpredictable, seemingly random fashion.

As might be expected, the improved spawning performance of New is not without some undesirable baggage. Recall that the first `multiply` thread is forked off by `main`, not by the `collector`. After spawning the thread, `main` does a join and switches out, waiting for its children to finish. As was noted earlier, once the new kernel's scheduler gets around to forking off the `multiply` threads, it does so more quickly and with less flutter than does Old's. But Figure 3-3 shows that New does not do this until after giving the `collector` an execution time-slice of over six seconds. Old, in contrast, forks the first

Figure 3-5: Improved Kernel: Zoom view of matrix multiplication

multiply thread after about four-fifths of second, negating the spawning advantage New has over it. So it seems that while the new kernel spawns threads more efficiently than the old it occasionally delays the actual creation of threads because its scheduler does not de-schedule running threads quickly enough. Figure 3-5 shows part of the execution of the matrix multiply computation on a kernel with an improved thread priority evaluation policy. As can be seen, the improved kernel reduces the amount of time given to the collector before the multiply is first allowed to run by about three seconds.

It is plain from these views that although New's scheduler has advantages over Old's scheduler, it can be improved even further. To the designer of the new scheduler[4], the views like those shown here confirmed that the changes perform as he specified. New's problem of thread's being descheduled for long periods is primarily due to a starvation bug which causes threads to occasionally get stuck at low scheduling priorities. The improved kernel shown in Figure 3-5 also fixes the starvation bug that is present in New. Using a progressive scheduling algorithm similar to the one in New, the improved kernel time shares the collector with the multiply threads instead of switching it out for several seconds. The curious problem of context-switch flutter detected in these views was unexpected and will be addressed and monitored in future tests.

Before moving on to our next example, it is important to note the utility of these PIE visualizations. Being able to see scheduling activity as these figures depict it is excellent assistance in analyzing and verifying the performance of scheduling functions. This kind of analysis of scheduling would not have been possible without the kind of performance visualization provided by PIE. These schedulers ran on a uniprocessor machine, however, and the central aim of PIE is support for the development and analysis of parallel computations. We now go onto an example dealing with the parallel execution of the matrix computations.

[4]David Black, of the Mach project, was the principal architect of the scheduling changes that differentiates the old, new and improved kernels.

3.4. Saturating a Multiprocessor

Multi-threaded computations like the matrix multiplier execute differently on multiprocessors than on uniprocessors. Obviously, a fundamental source of the difference lies in how many processors the threads time-share. On a uniprocessor, only one thread may run at a time, each waiting its turn as decided by the scheduler. In spite of anything a scheduler might do, however, the single processor holds the latent parallelism of a multi-threaded computation in abeyance. Although all multiprocessors support some form of parallelism, often it is not easy to match the parallelism demanded by a computation with that available on a particular system. Loosely-coupled and tightly-coupled machines, for example, efficiently support different kinds of parallelism. The architecture of a system is not the only factor affecting available parallelism. It is not remarkable that an otherwise desirable target system may have several users running computations simultaneously, encroaching upon the parallelism available to each of them. The performance of a multi-threaded computation thus depends as much on how it is scheduled as on how the user designs it.

Figure 3-6: Matrix Multiplier: First computation

Figures 3-6 and 3-7 are PIEscope views showing parts of two matrix multiplier computations executing simultaneously on a 16-processor shared-bus computer. There is much happening in these figures but our interpretation of them is restricted to the kernel's behavior; specifically, how the kernel's scheduler reacts to the periods when the two parallel computations together spawn many more threads than there are processors available to run them. Although our discussion contains some fairly detailed analysis, it is not intended as an exhaustive critique of the scheduler's performance. Rather, we mean it only as an another illustration of how a scheduler's behavior is observable using the PIE environment.

Figure 3-7: Matrix Multiplier: Second computation

3.4.1. Interpreting the Figures

As in the previous examples, the computations are matrix multipliers, but here they execute on a 16-processor shared-bus computer. As before, the colored rectangles represent periods when threads are running while white rectangles represent the periods when threads are switched out. Because the views depict executions on sixteen processors, cutting a swath vertically through a view at any point may slice through as many as 16 *running* threads instead of just one as was the case in the uniprocessor executions. The kernel schedules the threads using a simple, non-discriminating algorithm with a goal of assigning equal time-slices to competing threads. To facilitate the comparison of the views, the time-stamps of the computations are normalized to each other. That is, a time-stamp of 26.300 seconds in one computation, for example, is simultaneous with a time-stamp of 26.300 seconds in the other. The time stamps at the beginning and end of the PIE views in Figures 3-6 and 3-7 are almost identical[5], indicating that the views are snapshots of roughly the same period in time. A similar pairing of time stamps is done in Figures 3-8 and 3-9.

The two computations spawn several threads expecting that the scheduler will run them in parallel. By the time they terminate, each will have created 33 *logical* threads, but only 17 *physical* threads. In PIE, invisible user level run-time parameters limit the number of physical threads each computation may

[5]Current implementations of PIEscope do not support automatic time alignment of separate windows. The time limits in the views were set interactively using a mouse pointer and thus were difficult to align precisely.

execute in parallel based on considerations such as the architecture of the target system. Thus even though the computations may ask for 33 logical threads in parallel, they receive only 17 physical ones from the kernel. As will be shown shortly, the surplus logical threads must wait to execute until a physical thread becomes available (upon the termination of another logical thread, for example).

3.4.2. General Scheduling Behavior

Our discussion is restricted to the periods when the matrices are actually multiplied since these periods exhibit the most interesting behavior. The vertical metering lines in Figures 3-6 and 3-7 mark the approximate moment when the second computation commences multiplying its matrices. It is quickly seen in both views that especially in the later stages of the computations some threads run without ever being switched out. Other threads occasionally suffer some context-switch flutter, as clearly shown by the scattered periods of dense "squiggles," or are switched out for long periods before being allowed to run, as in the case of thread number 13 in the first computation in Figure 3-6. Unfortunately several of these scheduling phenomena not are explainable without intimate knowledge of the kernel's scheduling policy and what other tasks, if any, are running on the machine. In the discussion that follows, however, it is important to know only a few basic characteristics of the kernel. First, on this particular machine, the kernel designates one processor as a *master* processor on which all thread creation and other special work, such as certain disk accesses, is done. If all processors have been allocated and a new thread is spawned, the scheduler occasionally will not run the thread right away, apparently waiting to see if another thread finishes before adding the thread to the run-queue. Also, if at any time there are more threads than processors eligible to run, the scheduler will usually first context-switch threads that have been executing for long times. Knowing only these characteristics as well as the approximate number of threads vying for processors, we will be able to discern the cause of several of the more general phenomena like the context-switch thrashing occurring immediately after the PIE metering line in Figure 3-6.

Let's begin examining some general features by looking at the period immediately after the computation in the top view of Figure 3-8 starts spawning its multiply threads. Here, for about a second or two, several of the threads are executing unencumbered by context-switches, while others are either switched out, the `collector` and the number 13 `multiply` thread for example, or switching repeatedly as in the case of threads 14 and 15. For the most of this time, the first computation is executing fourteen threads while the bottom view in Figure 3-8 shows the second computation executing two threads, its own `main` and `collector`. Together, the two computations saturate the sixteen processors of the computer. As more processors are given work, it becomes more difficult for the scheduler to make equitable assignments. As shown in the top view, thread 13 of the first computation actually does run for a brief time, on the master processor, but the scheduler immediately removes it to run on another processor because there is probably some work queued up for the master. Each of the other processors is occupied, however, so the scheduler hangs the thread. Whether or not the scheduler is doing a good job here is a question we address later.

Back to the computations. We are examining the point immediately after the first computation spawns its multipiers when there is a total of roughly sixteen threads actively vying to run on sixteen processors. Shortly, however, the scheduler resumes running `collector` and thread 13 of the first computation. Here, instead of switching another thread out for a long period, the scheduler iterates through the threads, switching one or two out at a time, keeping the number of running threads at sixteen. This decision causes more frequent context-switching among the other threads, but might be a more equitable way to

Figure 3-8:

A closer look at the multipliers
Top: First computation
Bottom: Second computation

assign processors if, for example, the threads frequently communicate among themselves. As shown in Figure 3-8, the first computation's initial group of `multiply` threads finish executing before the second computation spawns its multipliers. Since the second computation has not begun vigorously competing for processors, the scheduler quickly begins assigning a new group of the first computation's multipliers to processors. Although the logical threads associated with these new multipliers were created about the

same time as the initial group of multipliers, each has been waiting all this time for one of the original logical `multiply` threads to relinquish the physical thread it is tied to. Not many of the first computation's physical threads are relinquished and reassigned because several of the original multipliers are parents refusing to relinquish their physical threads until their respective children terminate. Retaining their physical threads, several of the original multipliers switch out here. Later, they briefly switch back in once or twice to join a child or to exit themselves. The parents' reluctance to relinquish their physical threads until after all respective children have been joined coupled with the run-time thread limits can potentially lead to a form of dead-lock where no new threads can be created. These PIE views led to a modification of both the program and the run-time support code in order to permit as much parallelism as the machine could provide.

Looking at Figure 3-8 it is seen that shortly after the first computation's new group of multipliers begins, the second computation suddenly starts spawning the physical threads of its own multipliers (just after the first metering lines in top and bottom views) until it reaches the limit of 17 physical threads. It can be seen in the bottom view that several of the second computation's threads are not immediately assigned a processor. Rather, they are switched out for rather long periods before they commence running. Despite this, there are still more threads vying for processors than are available to run them. At this point the scheduler begins imposing a context-switching toll on the first computation, leaving a number of the second computation's threads alone. After a while, however, both computations begin to thrash quite heavily just after the second metering line in both views in Figure 3-8. The term "thrash" is appropriate because it calls to mind the thrashing that occurs in virtual memory systems when there are too many users demanding too many pages. Such scenarios often confound paging algorithms and cause them to repeatedly swap pages to and from disk. The heavy context-switching occurring here is consists of three dominant occurences: 1) switching reminiscent of the "flutter" described in the uniprocessor executions, 2) interprocessor exchanges of threads where two processors simply swap the threads they are running and, 3) "genuine" descheduling where a thread is switched out for a period of time. Eventually, the number of threads contesting for the processors diminishes and the scheduler is able to assign the threads without thrashing them.

PIE supports an execution view, called the *cpu-view*, that shows the behavior of processors during the execution of a computation. Figure 3-9 shows cpu-views of the first and second computations during roughly the same block of time as shown in Figure 3-8. As in the previous views, time is measured in milliseconds on the horizontal but here the machine's processors are ordered and arbitrarily numbered on the vertical. Opposite each cpu are alternating dark and white rectangles. A dark rectangle represents an identifiable executing thread while a white rectangle is a period when none of the respective computation's threads are running on the associated cpu. Each figure shows only those processors used by one of the computations. Only processor actions corresponding to threads belonging to the first computation are shown in the top view of Figure 3-9, for example. Examining the distribution of dark rectangles in both views, it is immediately obvious that before the second computation begins spawning its multipliers, the first computation has almost exclusive use of the 16 processors on the machine. After the spawning, however, the respective processor views show that at many times, only a fraction of the processors are allocated to each computation illustrating their competition for those resources.

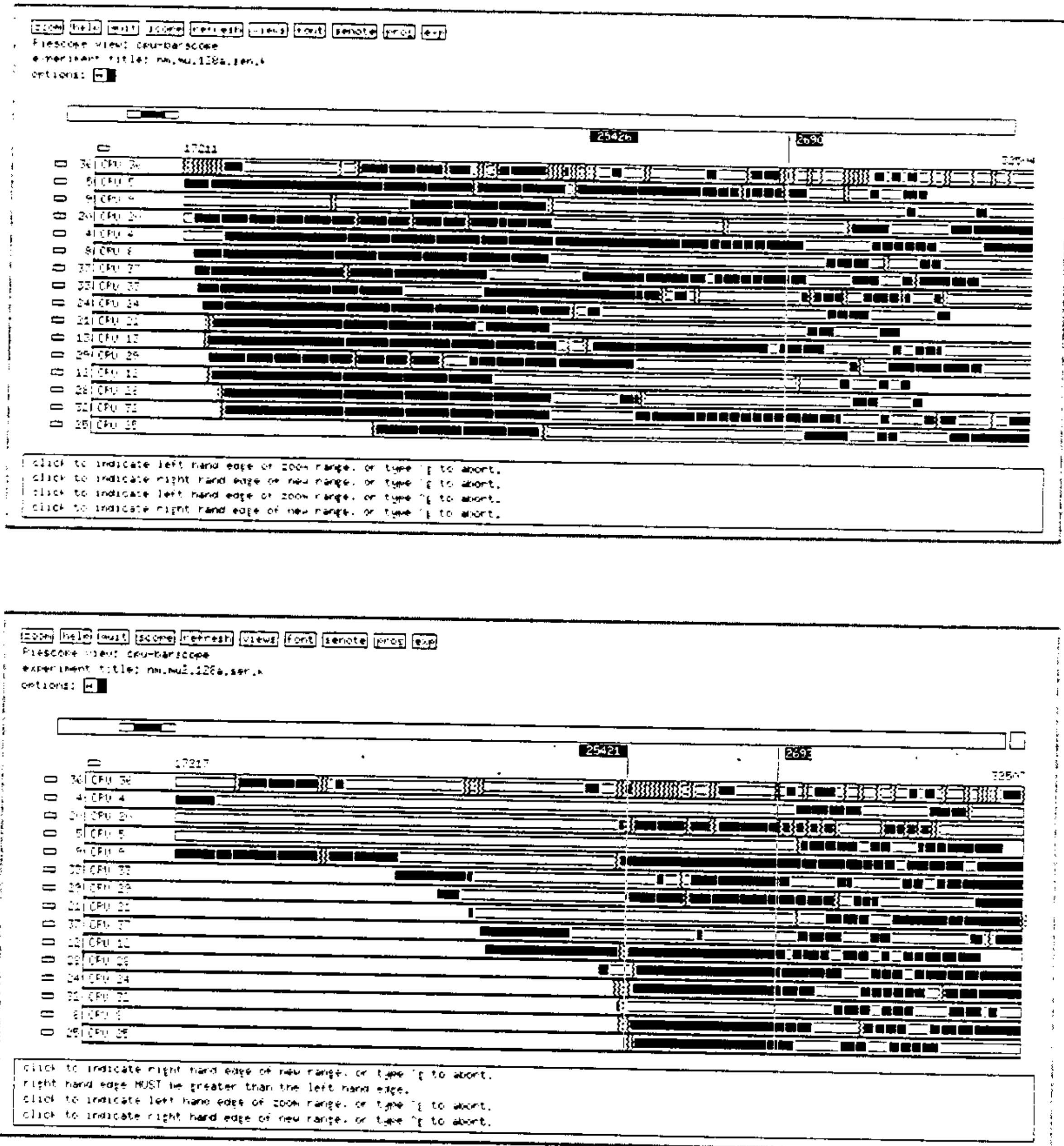


Figure 3-9:

Selected cpu views of the computation
Top: First computation
Bottom: Second computation

3.4.3. The scheduler's performance

Did the kernel do a "good" job scheduling these competing computations? The answer is probably that it could have done better. How to improve its performance is a difficult issue and is the subject of considerable research. See [12] [3] [14] [4] [17] [13] for examples of some of the issues. It is not always clear what the goals of a multiprocessor scheduler should be. Should it strive for high performance of isolated computations or high processor utilization for multi-programming support? The demands of parallel computations often conflict when they are executed on the same parallel machine. For example,

let's say there is a computation consisting of several sets of threads that, because of communication dependencies, must be scheduled as sets is the desired speedup is to be obtained. A scheduler whose first priority is to keep all its processors busy may never schedule those threads appropriately, especially if the machine is loaded. In the case of the two simultaneously executing computations just discussed, there were no special requirements for scheduling them. This was fortunate because the kernel had no way of handling special scheduling requirements of computations.

These PIE examples have shown the value of visualization in evaluating the performance of user-level computations and kernel behavior as well. We now conclude our discussion of performance visualization by remarking on some of the issues involved in correctly presenting visual information.

4. Improving the Environment

PIE is not just an environment for debugging performance. Underlying the architecture of the PIE system, is a theoretical framework for predicting, detecting and avoiding performance degradation. The basic ideas of this framework, enunciated in [16] [9] [20] and [19] are not rehashed here. Instead we discuss some of the issues of suitably retrieving and presenting performance information.

The experience of designing PIE has shaped our ideas on how to detect and isolate performance problems in a performance debugging environment. First, there is a question of whether there is a preferred methodology for monitoring performance in a visualization environment. Is there a best way to use performance debugging environments like PIE? What environment features do frequent users ask for? How must performance data be filtered and presented in order to accomodate these requests? Secondly, there is question of reporting accurate performance information. How does collecting performance data affect computations and what can be done to diminish the effects? Finally, the demands on an environment are different if performance data is collected only while developing a computation or whether data collection continues as an ancillary but permanent part of a computation after it is released for use.

4.1. Performance Monitoring Scenarios

Monitoring scenarios can be roughly divided into two broad categories: *1*) design phase monitoring and, *2*) design and deployment phase monitoring. In the first case, designers use performance debugging to sculpt and tune performance of computations before deploying them, *unmonitored*, for production uses. In the second case, computations are monitored not only during their design, but also after they have been deployed for use. The two cases place different demands on their respective monitors. For example, if a computation is to be released unmonitored, the most important characteristic of the programming environment is that it report the computation's performance as if monitoring perturbation were absent. It is occasionally desirable, however, to include monitoring as part of a deployed computation. For a computation that is monitored while it is deployed, it is more important that any performance penalty caused by the presence of a monitor is acceptable and that the monitor maintain a subservient, low-priority role with respect to the primary functions of the computation.

4.2. User Control of the Environment: Handling Changing Demands

Programmers usually do not know apriori where performance problems will arise. Realizing this ignorance, a programmer might begin measuring performance by monitoring only the high-level behavior of a computation. As the environment reports what the computation is doing, the programmer might decide to direct attention to more detailed behavior. For short computations, this step-by-step analysis may be spread over several successive executions, each one monitored differently. This is impractical for continuously executing computations (industrial control programs, for example) or those which require such a long time to execute that the turn-around time until the next observable execution is unacceptable.

4.2.1. Interactive Run-time Capabilities

Environments with interactive run-time capabilities let programmers probe computations as they execute. They allow programmers to peruse the execution of computations, moving their attention to different parts or asking for greater detail as the computations proceed through the different stages of their execution. For example, a programmer may initially wish to know only how long it takes the iterations of some loop to

execute. After examining the iterations for a while, the programmer may decide either to probe deeper into the loop or divert his or her attention to some other part of the computation ... all during run-time. In order for an environment to be able to adapt to the run-time vicissitudes of a programmer it must have an interactive interface. More ambitious environments might support dynamic changes to the object code of a computation while it executes so that programmer can conveniently test coding changes during run-time instead of resorting to time consuming recompilation [2]. The PIE environment allows programmers to observe computations while they execute but has only limited interactive features. These capabilities are being extended to support more versatile interaction with executing computations.

4.2.2. Tailoring a Monitor to a Computation

In cases where computations are monitored after having been deployed, programmers might be interested in adjusting monitoring characteristics in hope of improving its performance and thus the performance of the entire computation. A programmer might decide, for example, that monitoring uses more I/O bandwidth in an application than it ought to. Because the monitor is supplied by the programming environment, however, programmers can not modify the monitor as freely as they would computations without risking the integrity of the montoring code. In order to assist programmers in improving monitoring performance, environments should carefully provide means by which designers can tailor the characteristics of the monitor in order to better meet the performance requirements of their computations. The environment, however, must restrict the kinds of changes that are permitted so as to ensure that programmers can not introduce any bugs into the monitor. When tailoring a monitor to the performance goals of a computation, programmers have to balance the demands of the computations against how much and what kind of information is desired from a monitor. If capturing every selected event is important, programmers must estimate parameters like the maximum rate at which the monitor has to record events. Currently, PIE permits a programmer to modify only a few selected monitoring parameters. As part of the effort to improve the interactive nature of PIE, however, more options are promised for customizing it's monitoring system.

4.3. Perturbing a Computation

Performance monitoring which significantly alters the component execution times of a computation and reports only the corrupted times back to programmers compromises their ability to discern performance bottlenecks. Consistently and grossly corrupted performance information can be useless or even harmful in attempting to improve a computation. Monitors corrupt performance measurements by competing with computations for hardware and software resources like processors, co-processors, schedulers and I/O drivers. The principal measure of this competition is *time*. When a monitor takes time away from a computation ... when it delays or lengthens a computation ... it *perturbs* the computation. Time penalties are only "first order" perturbations. Lengthening the execution times of the individual components of a computation might eventually distort the order of parallel events and change what kind and how much work the computation does. Direct distortion of event order and general computation behavior via artificial sequentialization, for example, is a risk because monitors communicate with computations.

4.3.1. Measuring Perturbation

Unfortunately, monitoring run-time performance almost always perturbs a computation. A problem confronting programming environments like PIE is how to report accurate performance information despite monitoring perturbation. Since monitoring perturbations cannot be completely *eliminated*, they must be compensated for by *measuring* them in order to be able estimate how the computations would

perform without the perturbations. Perturbation measurements consist of calculating where and how much a monitor slows a particular computation, whether the monitor artificially sequentializes the computation and whether the workload is affected.

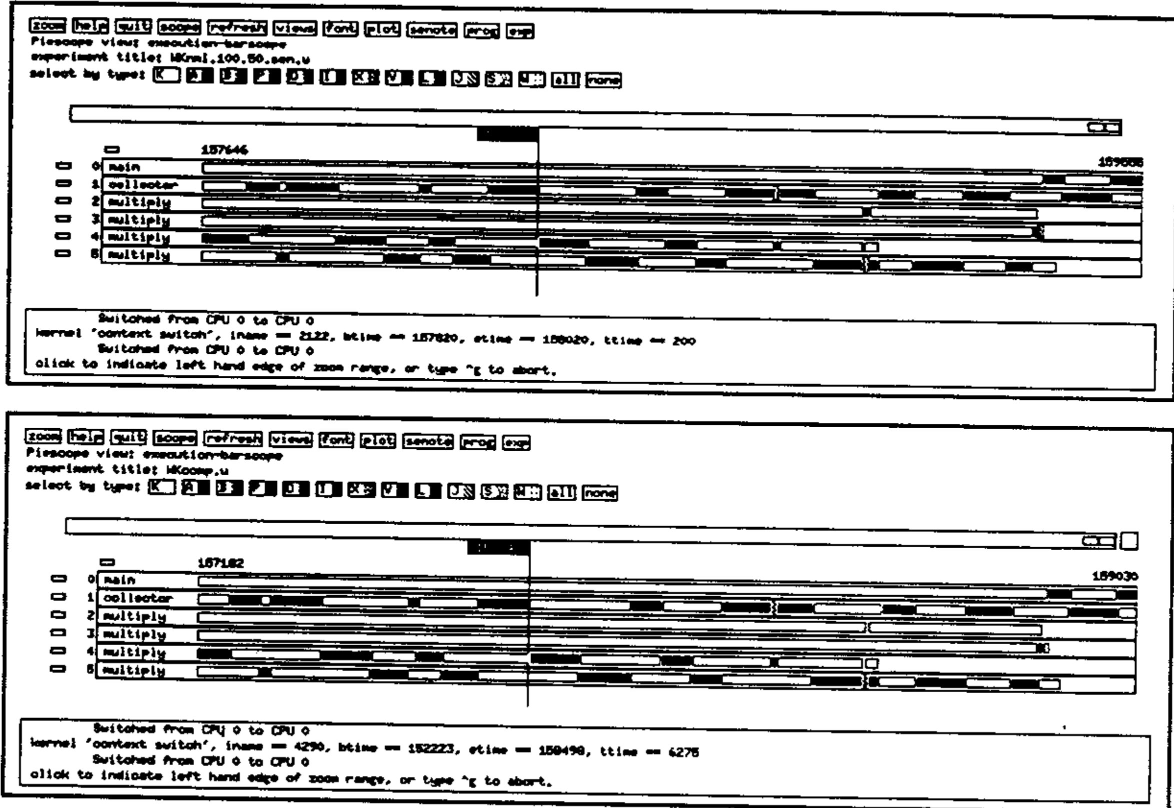


Figure 4-1: Uncompensated and Compensated Views of Part a Matrix Multiply Executed on a microVax II (XF29 Kernel)

4.3.2. Measurement Based Compensation for Perturbation

Currently, PIE attempts to compensate for perturbation from the sensor firings on micro-Vaxes and other uniprocessors using a fairly simple compensation algorithm where the timestamp of each event is adjusted in accordance with how many have events occurred before it. The algorithm compensates for each event by subtracting from its time stamp a quantity corresponding to the sum of all the sensor firing times of previous events. The top view in Figure 4-1, for example, shows the tail end of a matrix multiplication executing on a microVax II running an XF29 kernel (the old kernel in the uniprocess scheduling example) . The bottom view in the figure shows the same computation after compensation by PIE. As can be seen, the computations appear identical except that their termination times and the metering line point to their differing execution times. Even without compensating for the sensor firings, the execution times of computations executing on micro-Vaxes with and without kernel monitoring are generally within one percent of each other. Although the algorithm is fairly successful at compensating uniprocessor executions, it does not attempt to modify the total number of context-switches counted in a computation even though it is reasonable to expect that shorter computations should suffer fewer context switches.

There is risk of reordering of events using such a simple compensation algorithm. Although this is a danger in compensating uniprocessor executions, this inadequacy is particularly severe for

multiprocessor executions. More than just sensor firing times need to be taken into consideration before a realistic adjustment of the views can be done. Certain types of information about the computation, synchronization points arising from communication and thread creation for example, is required if compensation is to be attempted. In addition to the computation information, knowledge is required about monitor parameters such as sensor firing times, points where the monitor may introduce synchronization overhead and how monitor threads are scheduled with the computation. By knowing firing times, the relative delay caused by monitor sensors can be determined at any time in the computation. By knowing when and where the monitor and computation are scheduled, any significant time-slices when the monitor preempted the computation can be accounted for. Because monitors and computations might compete for resources other than processors, such as I/O bandwidth, the monitor's use of these resources should be known. PIE is investigating how these these factors affect compensation techniques as well as how to detect unrecoverable perturbation.

4.4. Monitor Priorities

Which is more important, the computation's execution or the information about it? This question does not have a single answer for it depends on whether the computation is under development or in use. In either case, not all information about the computation is interesting, and what is interesting information may change as the computation progresses.

4.4.1. Measuring the Entire Computation ...

When developing a computation, a programmer wants to ensure that it performs correctly and adequately. During the run-time observation of a computation, the designer is usually examining its behavior quite closely. In the initial development stages it is important that very little performance data, if any, is lost by the monitor. Failure to record performance data can happen if, for example, buffers for caching performance information fill before they can be read. A programmer who is examining the synchronization behavior of a computation would be ill served if a monitor begins losing events when synchronization occurs. Environments must guarantee that every selected event is recorded, otherwise the ability of programmers to discern what their computations really do is compromised. To do so, it may be necessary for monitors to occasionally pause computations in order to flush a number of previously recorded events to monitor files before resuming them.

4.4.2. ... versus Maintaining a Performance Level

The perturbation of a monitor that is part of both the design and deployment phases of a computation is no longer perturbation per se, but merely part of the overall behavior of the computation. The real behavior of the computation is that of both the monitor *and* the primary functions of the rest of the computation. In most cases, performance data about a computation does not need to be adjusted for the monitor's perturbation. Indeed, what is important is not monitoring perturbation, but monitoring performance. In such cases, the most important requirement of a monitor is that it never pause a computation using high-level synchronization barriers. Such barriers risk delaying the execution of important operations. For example, a monitor used in an airflight control application would be used not only to ensure the computation's integrity but also as a means to retrieve information about what the flight controller and computation were doing when a pathological condition arises (a plane crash, for example). During normal operation it is important that the monitor interfere as little as possible with the computation, but it would be highly undesirable, for example, if the monitor stopped the computation (and, therefore, the flight controller) in order to "catch up" on recording events. No matter how unappetizing losing events may be it is more important that the computation continue unperturbed.

5. Conclusions

Visualizing the behavior of computations, as is done in PIE, is a suitable way to study and improve their performance. Although PIE is a successful, useful research effort aimed at assisting programmers in developing better computations by visually matching performance data onto programming constructs, it is not limited to merely improving computations. It is also a valuable educational tool for revealing the complexities of both sequential and parallel programming. It helps to remove the veil of mystery surrounding the execution of computations on complex systems. Even an operating system's scheduling behavior, a seemingly esoteric activity over which the programmer frequently feels and is powerless, is revealed simply and elegantly in PIE. PIE is currently available for most workstations and tightly coupled parallel computers running Mach. It supports programming in C, MPC, C-threads [5], Fortran and Ada. As we complete the work described earlier, the power and value of visualization offered by PIE will help to remove additional shrouds of complexity from sequential and parallel computations and the systems that run them.

6. Acknowledgements

David Black, of the Mach Project, provided us with expert advise during our development of kernel monitoring. Mark Russinovich and Glenn Schuster have been helpful in improving the run-time and graphics capabilities of PIE.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Bolub, R. Rashid, A. Tevanian, M. Young.
Mach: A New Kernel Foundation for Unix Development.
In *Proceedings of USENIX 1986 Summer Conference*, pages 93 - 112. Computer Science Department, Carnegie Mellon University, Summer, 1986.

[2] Ziya Aral, Ilya Gertner.
Non-Intrusive and Interactive Profiling in Parasight.
Technical Report ETR 88-006, Encore Computer Corporation , 1987.

[3] Shahid H. Bokhari.
Partitioning Problmes in Parallel, Pipelined, and Distributed Computing.
IEEE Transactions on Computers 37(1):48 - 57, January, 1988.

[4] Timonth C.K. Chou, Jacob A. Abraham.
Distributed Control of Computer Systems.
IEEE Transactions on Computers C-35(6):564 - 567, June, 1986.

[5] Eric C. Cooper, Richard P. Draves.
C Threads.
Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June, 1988.

[6] Susan B. Dart, Robert J. Ellison, Peter H. Feiler and A Nico Habermann.
Software Development Environments.
IEEE Computer 20(11):18 - 28, November, 1987.

[7] R. Fitzgerald, R. F. Rashid.
The Integration of Virtual Memory Management and Interprocess Communication in Accent.
ACM Transactions on Computer Systems 4(2), May , 1986.

[8] Daniel D. Gajski, Jih-Kwon Peir.
Essential Issues in Multiprocessor Systems.
IEEE Computer 18(6):9 - 27, June, 1985.

[9] Francesco Gregoretti, Zary Segall.
Programming for Observability Support in a Parallel Programming Environment.
Technical Report CMU-CS-85-176, Computer Science Department, Carnegie Mellon University, November, 1985.

[10] M. B. Jones, R. F. Rashid, M. Thompson.
MatchMaker: An Interprocess Specification Language.
In ACM (editor), *ACM Conference on Principles of Programming Languages*. Computer Science Department, January, 1985.

[11] Ted Lehr, David Black, Zary Segall, Dalibor Vrsalovic.
MKM: Mach Kernel Monitor, Description, Examples and Measurements.
Technical Report CMU-CS-89-131, Department of Electrical and Computer Engineering and the School of Computer Science, Carnegie Mellon Univerity, April, 1989.

[12] Virginia Mary Lo.
Heuristic Algorithms for Task Assignment in Distributed Systems.
IEEE Transactions on Computers 37(11):1384 - 1397, November, 1988.

[13] John K. Ousterhout.
Scheduling Techniques for Concurrent Systems.
In IEEE Computer Society (editor), *The 3rd International Conference on Distributed Computing Systems*, pages 22 - 30. Electrical Engineering and Computer Sciences, University of California at Berkeley, October, 1982.

[14] Constantine D. Polychronopoulos, David J. Kuck.
Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers.
IEEE Transactions on Computers 36(12):1425 - 1439, December, 1987.

[15] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky and Jonathan Chew.
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
IEEE Transactions on Computers 37(8), August , 1988.

[16] Zary Segall, Larry Rudolph.
PIE - A Programming and Instrumentation Environment for Parallel Processing.
Technical Report CMU-CS-85-128, Computer Science Department, Carnegie Mellon University, April, 1985.

[17] J.A. Stankovic.
An Application of Bayesian Decison Theory to Decentralized control of Job Scheduling.
IEEE Transactions on Computers C-34(2):117 - 130, February, 1985.

[18] C.R. Vick, C.V. Ramamoorthy, S.M. Jacobs (chapt. 28).
Electrical/Computer Science and Engineering Series. Handbook of Software Engineering.
Van Nostrand Reinhold, 135 West 50th St., New York, N.Y. 10020, 1984, pages 234 - 246, Chapter 11: Operating Systems.

[19] D. Vrsalovic, Z. Segall, d. Siewiorek, F. Gregoretti, E.Caplan, C. Fineman, S.Kravitz, T. Lehr, M. Russinovich.
Performance Efficient Parallel Programming in MPC.
Technical Report CMU-CS-88-167, School of Computer Science, Carnegie Mellon University, July, 1988.

[20] D. Vrsalovic, D. Siewiorek, Z. Segall, E. Gehringer.
Performance Prediction and Calibration for a Class of Multiprocessor Systems.
IEEE Transactions on Computers 37(11):1353 - 1366, November, 1988.

[21] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, R. Baron.
The Duality of Memory and communication in the Implementation of a Multiprocessor Operating System.
In *Proceedings of the Symposium on Operating System Principles.* School of Computer Science, Carnegie Mellon University, November, 1987.