INTEGRATED VISUALIZATION OF PARALLEL PROGRAM PERFORMANCE DATA

KAREN L. KARAVANIC*, JUSSI MYLLYMAKI*, MIRON LIVNY*, AND BARTON P. MILLER*

Abstract. Performance tuning a parallel application involves integrating performance data from many components of the system, including the message passing library, performance monitoring tool, resource manager, operating system, and the application itself. The current practice of visualizing these data streams using a separate, customized tool for each source is inconvenient from a usability perspective, and there is no easy way to visualize the data in an integrated fashion. We demonstrate a solution to this problem using Devise, a generic visualization tool which is designed to allow an arbitrary number of different but related data streams to be integrated and explored visually in a flexible manner. We display data emanating from a variety of sources side by side in three case studies. First we interface the Paradyn Parallel Performance Tool and Devise, using two simple data export modules and Paradyn's simple visualization interface. We show several Devise/Paradyn visualizations which are useful for performance tuning parallel codes, and which incorporate data from Unix utilities and application output. Next we describe the visualization of trace data from a parallel application running in a Condor cluster of workstations. Finally we demonstrate the utility of Devise visualizations in a study of Condor cluster activity.

Keywords: data visualization, parallel programming, performance profiling

1. Introduction. Visualization of parallel program performance data allows one to graphically explore the events that occur during the execution of these programs. Visualization provides an intuitive way to debug parallel programs and monitor their performance, and it can reveal aspects of the execution that could not be spotted by simply browsing performance data files textually.

During the execution of a parallel program, performance data may be generated by many components of the system, including the message passing library (such as PVM or MPI), a monitoring tool (such as Paradyn), a resource manager (such as Condor), the operating system, and the application itself. The current practice for visualizing these data streams is to use a separate, customized tool for each source. This is inconvenient from a usability perspective because the user is forced to learn how to use several visualization tools. What is more problematic, however, is that there is no easy way to visualize data from different sources in an integrated fashion. The ability to explore the detailed data of all components participating in the execution of the program in a unified manner could lead to better understanding of how the components interact and work together.

In this paper we describe a generic visualization tool called Devise [2, 6] which is designed to allow an arbitrary number of different but related data streams to be integrated and explored visually in a flexible manner. Devise can visualize performance data from individual parallel program runs just as well as a custom-built parallel program visualization tool can, but its ability to display data emanating from a variety of sources side by side provides a significant advantage. Devise also allows data from multiple executions to be juxtaposed and compared.

The key to successful visual integration is for each data source to have a well-defined data export format that is compatible with the formats of other data sources. The schema mechanism in Devise allows much of the mapping and conflict resolution between dissimilar data sets to be performed on-the-fly. Still, "better data" yields

*Computer Sciences Department, University of Wisconsin–Madison

better visualizations. In this paper we describe how the Paradyn Parallel Performance Tool was interfaced with Devise to provide more accurate and intuitive visual information about parallel programs.

Devise is a very generic visualization tool in that it can be used in any domain requiring the visualization of record-oriented data sets. Other fields where we have used Devise include the financial markets, marketing, clinical medicine, biochemistry, and soil science. The visualization and integration efforts we describe in this paper are equally applicable to these fields as well.

2. Devise: A Data Exploration and Visualization System. Devise is a data exploration and visualization system designed to handle multiple, large data sets using off-the-shelf hardware with limited main memory sizes [2, 6, 1]. Data can be large in volume and complex in structure (multi-dimensional and/or hierarchical), and may be imported from a variety of sources such as disk and tape files, database servers, external programs, and World Wide Web resources. Devise allows the user to integrate a collection of heterogeneous data sets visually by linking their graphical attributes (e.g. location, color, and size). In this paper we show how Devise can be applied to visualize and integrate parallel program trace data. The generality of Devise has made it possible for us to apply the same visualization and integration principles in other domains such as financial markets, soil science, clinical medicine, and biochemistry.

2.1. Devise Data Model. Devise visualizes *streams* of data, that is, record-oriented data sets such as time series or trace files. Our focus on stream data allows Devise to provide a more flexible mapping mechanism to the user, and to incorporate performance optimizations specific to stream data. It also lets us efficiently interface Devise to record-based data servers such as SQL relational databases or a SEQ sequence database [12, 11].

Common to exploratory data analysis and trace file visualization is the need for a powerful browser. Many visualization systems assume that the data set being visualized will fit in main memory, or assume that virtual memory can be used as a safeguard when the data set is larger than main memory. Devise makes the data browsing aspects of visualization flexible and allows large data sets to be visualized without resorting to virtual memory. The flexibility we offer to users comes partly from the ability to dynamically define mappings from data to graphical attributes. The performance limitations of virtual memory are to a large extent bypassed with the design of the buffer manager and the query processor in the Devise system and with the mechanisms for caching data in alternate forms. These features make it possible to effectively visualize and explore large quantities of data with limited main memory sizes.

2.2. Devise Visualization Model. In Devise, visualization is structured into a five-stage process, as shown in Figure 1. At the source of the data flow is a *data stream* which consists of binary or ASCII records. Each record contains one or more *attributes* whose type and order is described by a *schema*. The combination of a data stream and a schema makes up *TData*.

Data streams that are not in record format or whose record structure cannot be captured with the schema facility can still be visualized via conversion utilities. The conversion from the original format of the data stream to TData can be performed either as a post-processing step of the submitting application, or as a pre-processing step of Devise. A small extension function which parses a new data format and

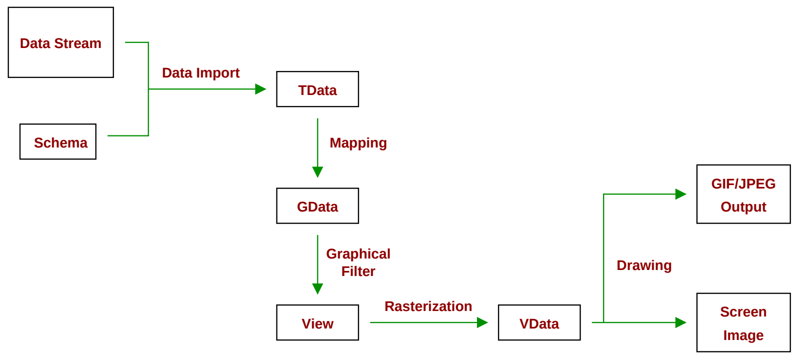

FIG. 1. *The Devise visualization model. Visualization is performed as a five-stage data transformation. The intermediate data forms can be materialized and cached. The cached data is reused whenever a user changes the visualization parameters, such as the graphical filter.*

converts it to record format at run-time can be linked with the Devise executable.

Each TData record is mapped to a *GData* record with a fixed schema of graphical attributes: *X*, *Y*, *size*, *color*, *pattern*, *orientation* and *shape*. Each shape (e.g. vector, rectangle, polygon, or oval) can have optional parameters specifying, for example, line width. A GData attribute is typically defined to take the value of a TData attribute or some expression of TData attributes. GData attributes can also assume constant values. The *mapping* of TData attributes to GData attributes in this way gives the user full control over the graphical representation down to the record level.

A *graphical filter* determines the portion of GData to be displayed by defining a query over the GData attributes. Scrolling and zooming are accomplished by defining a range query over the X and Y attributes in the filter. The subset of GData records satisfied by the graphical filter defines a *view*. A view has additional information relating to the presentation of the selected GData records, including axis, title, and background color information. The display of statistics on the selected GData records can also be enabled on a view basis.

The next step is the conversion of GData to a pixel image. The size of a view (number of pixels) plays a key role in determining the accuracy of the rasterization process and the resulting *VData*. Although the most important role of VData is to be painted on a screen and to be exported to other graphics formats, VData is also cached and used for further transformations. VData in many cases takes up less space than the subset of GData that defined it. To save CPU and data transfer time, it is desirable to be able to reuse VData via pixel manipulation when a view's graphical filter changes or when the user recalls a filter used earlier in the visualization session.

The final step is the placement of VData into windows (if running Devise interactively) or exporting it as a GIF or JPEG image (batch mode). A window provides the screen real estate, and the user can control the real estate by moving and resizing the windows. Various layouts are available for placing two or more VData into a window: a vertical, horizontal, tile, and a pile layout.

A *graphical link* is a key mechanism offered by Devise that makes data navigation easier by a user. A graphical link makes two or more views share some graphical attributes of their graphical filter. An X link, for instance, is used to display the same X range in all views sharing that link; zooming or scrolling in the X direction in any of the views zooms or scrolls the others as well. Or, linking two views by the color attribute and selecting only red shapes in one view (via the filter) would perform the same selection in the other view as well.

A *graphical cursor* associates two views in a different way. A user may select two distinct subsets of the same GData by using different graphical filters. For instance, one may have a *global view*, containing all GData records, while a *focus view* contains a small subset of the GData records. A cursor associates the two views so that when the VData are drawn into a window, a cursor symbol appears in the global view, indicating the X and/or Y ranges selected by the focus view. Moving a cursor in the global view with the mouse updates the graphical filter of the focus view to reflect the new location of the cursor. The typical application of this feature is that the user recognizes interesting trends or events in the global view, and moves the cursor to cover the interesting area, which is then displayed in more detail in the focus view. The size and location of the cursor symbol is determined by the zoom level and scroll position of the focus view.

3. Paradyn: A Tool for Scalable Parallel Performance Tuning. Paradyn [8] is a tool for measuring and understanding the performance of parallel and distributed programs. It consists of a data collection facility, an automated bottleneck search tool, and a data visualization interface.

Data collection in Paradyn uses dynamic instrumentation to provide detailed, flexible performance information without incurring the space (and time) overhead typically associated with trace-based tools. Instrumentation code is inserted, removed, and modified during application execution, recording performance information in counters and timers. These counters and timers are then periodically sampled, yielding accurate information with low perturbation. Instrumentation requests are made by specifying a *metric* and a *focus*. A metric is a time varying function that characterizes some aspect of a parallel program's performance. A focus is a list of program components that defines a particular point of interest, such as a particular procedure for a particular process.

The Performance Consultant is a tool that automates the search for a predefined set of performance bottlenecks. Potential bottlenecks are specified as a hierarchy of hypotheses. Each hypothesis includes a test definition that translates into specific instrumentation requests as an application is being measured. The resulting data collected is compared against a user-defined threshold to yield a determination of truth or falsity for each potential bottleneck.

Paradyn's visualization interface allows users to easily add new visualizations to the system. A simple library and remote procedure call interface allow external visualization processes to display Paradyn performance data in real time. This interface is used by the standard visualizations included with Paradyn, and is designed to allow other visualization tools to display Paradyn data. All external visualizations are listed in Paradyn's *visi* menu; the user simply selects a visi by name then chooses a set of foci and metrics for display. The selection of a list of performance metrics for a list of foci can most easily be pictured as a two-dimensional array. The visi library provides a C++ class, called the DataGrid, which is the visi programmer's interface to performance data. Each element of the DataGrid is a time histogram, representing the metric's time-varying behavior. The library also provides aggregation functions, such a minimum, maximum, current, average, and total, that can be invoked over each DataGrid element. Also included is an interface which allows access to the visualization interface calls via Tcl [9] commands.

4. Combining the Power of Paradyn and Devise. Devise can be used to develop visualizations spanning multiple program executions, allowing us to detect trends and patterns that might be difficult to see from a single run. Another useful feature is the ability to visualize data from distinct sources. We demonstrate visualizations with application data, output from the Paradyn Parallel Performance Tool, and operating system data generated with the Unix utilities `iostat` and `vmstat`. We have also successfully used Devise to visualize MPI [7] log files and PICL [3, 15] log files.

4.1. Exporting Paradyn Data. We created two new visualizations that generate both schema and data files ready to be imported into Devise. The actual data reported varies with the user's selection of metric/focus pairs. Generating the schema allows easy incorporation of dynamic performance data into a Devise session. To register the new visis with Paradyn we simply list them in the Paradyn setup file; no changes to Paradyn itself are required.

To scalably monitor large and long-running codes, Paradyn uses an internal data structure called a time histogram, which stores collected performance data in a fixed-size list of data buckets. Each bucket contains a value collected over a time interval of uniform length. Once all buckets have been filled, the histogram is "folded": the histogram bucket interval is doubled, and adjacent pairs of values are merged to fit the new bucket sizes. So, regardless of the length of time of a particular performance tuning session, the amount of space required to hold performance data remains fixed.

Our first visi, histoSaver, allows the user to save the current contents of selected time histograms to a file. The number of data points is equal to the number of valid histogram buckets, with a fixed maximum. The length of the time interval represented in each data value is the bucket width at the time the data was saved to the file.

Our second visi, deviseFeeder, is designed to allow Devise data visualization to occur in real time. New data values are written to the file as they are received by the visualization. There is no fixed maximum to the amount of data that may be generated by this approach. This visi was designed primarily for future use as a link to Devise for real-time visualization of performance data as it is being collected by Paradyn.

4.2. The Application. Moma.pvm [14] is a PVM message passing implementation of the 'moma' ocean model code, a version of the Bryan-Cox-Semtner code which was developed for use with array processor computers. The application follows a *master-workers* paradigm where there is a single master process which generates work steps to be computed, and a collection of worker processes. In moma.pvm, the ocean is split up into a number of sub-volumes, each the responsibility of a separate worker process. Each process carries out all calculations required for its volume of ocean. It also exchanges data on boundary points with processes responsible for neighboring areas of ocean. The sub-volumes are each made up of vertical columns of grid boxes and therefore can be specified by a two-dimensional horizontal array. The master process controls the model run, reads the input files and handles archiving. The master also spawns the worker processes which actually carry out the model calculations.

4.3. Examining Multiple Executions in a Single Tuning Session. In this section we describe a performance tuning session which examines several executions of the moma.pvm ocean modelling code. It is not uncommon for programmers to develop codes which will be used across different architectures, datasets, even message passing libraries. Visualizing performance results across changing environments can aid the programmer in detecting bottlenecks without changing a code based on results which may in fact be spurious.

Figure 2 shows performance data generated using Paradyn and our deviseFeeder visualization module, from three different executions of the ocean model. In runs A and C we measured application performance on four networked Sun SPARCstations running Solaris 2.4. In Run B we measured performance on our Cluster of Workstations (COW). The COW consists of 40 SPARCstation20 workstations, each of which contains two 66 Mhz Ross HyperSPARC processors, connected by a 10Mbit/s Ethernet. In all runs the application includes a total of one master process and four worker processes.

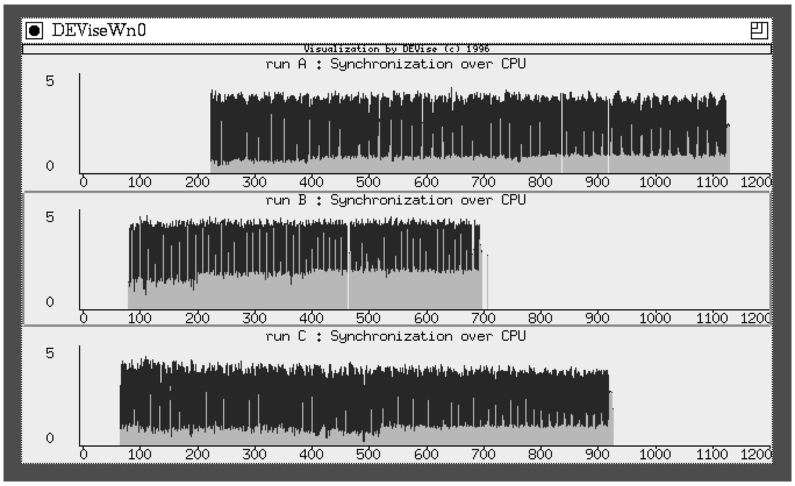

FIG. 2. *CPU versus Synchronization. CPU (the lighter segments) and Synchronization time (the darker segments) are displayed as split vertical bars for each data collection point. Three different application runs are shown.*

We display a bar graph with each bar length equal to the sum of synchronization waiting time plus CPU time. Times are summed across processes, so the maximum possible would be 5.0 for these runs. We produced one data stream per application run; each data stream record contains a start and end time, and data values for CPU time and Synchronization time. We link the X axes (wall clock time in seconds) of the three bar charts so we can scroll and compare data for identical timestamps. For each run we display data beginning after the run has started and ending when execution is complete. Aligning execution time in this manner allows us to see immediately the difference in total execution time across the runs: Run A = 1133 seconds; Run C = 930; Run B = 707. The speedup from Run A to Run C was accomplished by changing the configuration we used for the run: in Run A, we use a total of 4 workstations, which means a single workstation runs the application master process, one application worker process, plus the Paradyn front end process. For Run C, we moved the Paradyn front end process over to a separate workstation. The difference between these two runs and the COW run is attributable to differing machine speed.

The split bar graph also allows us to see at a glance the ratio of computation to synchronization in each execution. Run B achieves a ratio of roughly 1:2, while runs A and C give a much poorer result in the range of 1:3.

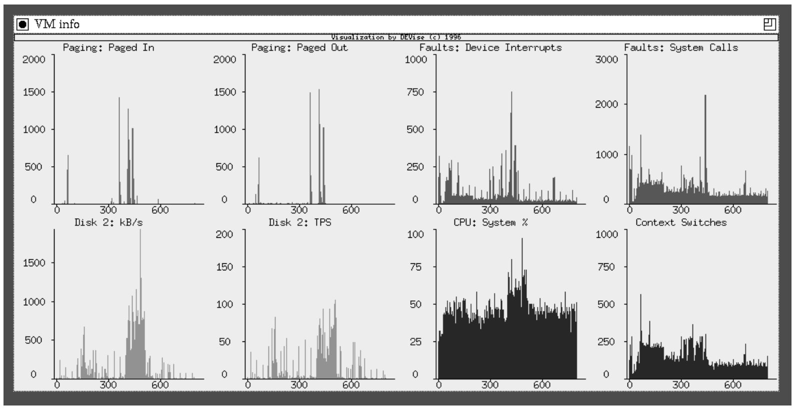

FIG. 3. *Visualizing Operating System Performance. The data for these graphs was collected using the Unix* `vmstat` *and* `iostat` *utilities. They show operating system statistics for one workstation during Run B.*

4.4. Visualizing Data from Multiple Sources. Performance tuning in general requires data from a variety of levels of an execution run. Operating system factors such as machine load, memory use, and network contention can all affect application performance. Figure 3 shows data generated using the Unix utilities `vmstat` and `iostat`. This information was collected during Run B of the ocean modelling code. Machine load is of particularly importance when performance tuning parallel codes running on workstations, where each node may be performing a set of varying additional computations in addition to the program being measured.

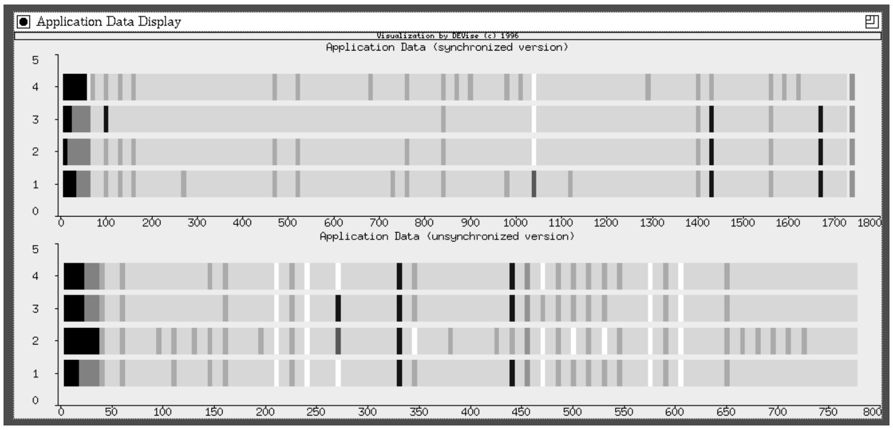

FIG. 4. *Visualizing Application Data for Two Executions. Each horizontal bar represents a single worker process. The colors represent distinct locations within the computation. To check for synchronization, scan along a vertical slice.*

Devise's flexibility allows us to include results from application output in our tuning session in an intuitive manner. Figure 4 shows status information gathered during execution and generated directly by the ocean program for two different versions of the modelling code. The master process keeps track of where each worker process is in the computation cycle and periodically prints a status table containing this information. We wrote a simple Tcl script to reformat this output and created a data stream in Devise. Each record of the data stream contains a processor id, process position, and checkpoint time. Each horizontal bar in the display represents one worker process. Each possible computation location is assigned a different color code. So, scanning along a single horizontal line shows the progression of each process through various computation phases. Scanning the display along a particular vertical slice, all process colors will be identical if the processes are at the same computation step when the status check is done. So, a closely aligned computation would appear as vertical stripes of color, whereas a poorly aligned computation would result in broken patches of color along each vertical slice. We illustrate this with the two different program versions visualized here: the top run shows a version with extra synchronization of the worker processes; the bottom run shows the regular version. Although the processes are at the same location through most of both runs, we can clearly see more out-of-synch checkpoints in the bottom display.

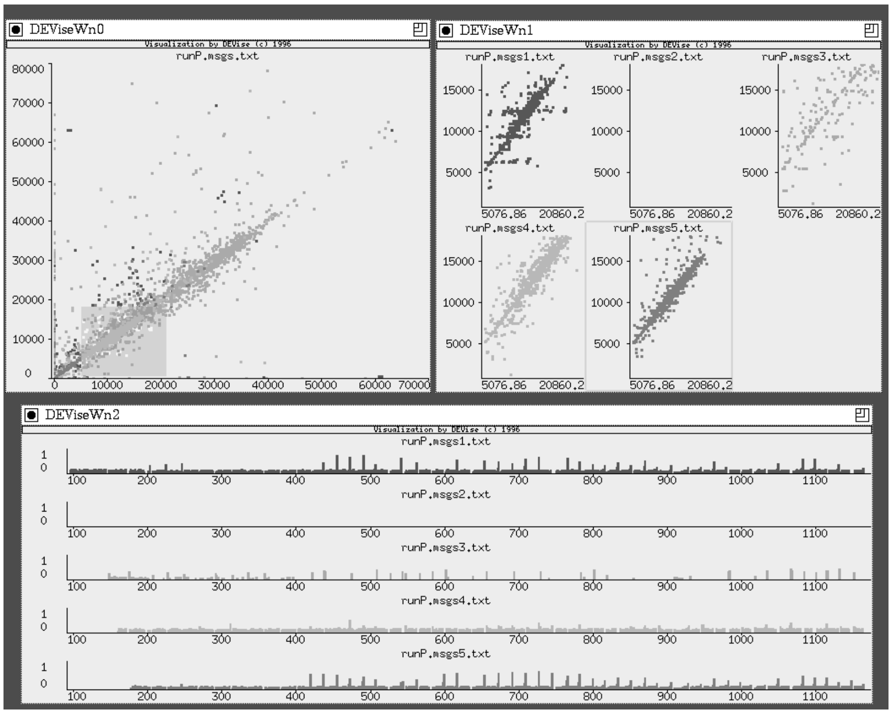

FIG. 5. *Visualizing Message Behavior. The scatter plot, in the upper left of the screen, contains points color-coded by process. The bar graphs in the lower display plot CPU utilization over time, one graph for each process. The detail graphs on the upper right of the screen show a scatter plot detail for each process.*

We capitalize on Devise's graphical cursor feature to construct a more complex visualization which enables us to study message passing behavior of the application. The result is shown in Figure 5. Using a scatter plot, we graph message bytes sent versus message bytes received for each of the five processes in an ocean modelling run. The scatter plot, in the upper left of the screen, contains points color-coded by process. The bar graphs in the lower display plot CPU utilization over time, one graph for each process. The detail graphs on the upper right of the screen show a scatter plot detail for each process, helpful for examining dense portions of the scatter plot. The graphical cursor box can be moved around the scatter plot with the mouse; the data which corresponds to the section contained within the cursor box is graphed in the two other displays shown. This powerful feature enables the user to examine outliers or unexpected data points and see where on the time line they occur.

To create this visualization we generated five data streams using Paradyn output. Each data stream record contains a start and end time, process id, message bytes sent, message bytes received, and CPU utilization. In the example shown, we can determine the overall communication pattern of the set of processes. A producer-consumer model would yield a single processor with message bytes sent dominating, and the rest dominated by message receives. Our ocean modelling code contains nearly balanced exchanges of data between all of the worker processes. A single worker process shows larger values overall for both sends and receives; this process, the first worker created, performs additional messaging for coordination. The master process, in the middle bar graph at the top of the breakout displays, actually performs no messaging.

5. Visualization of Condor Application Traces. Next we describe the visualization of trace data from a parallel application running in a Condor cluster of workstations. Condor is a distributed resource management system for large heterogeneous clusters of workstations [5]. Its design has been motivated by the needs of users who would like to use the unutilized capacity of such clusters for their long-running, computation-intensive jobs. Condor schedules batch jobs on idle workstations within the cluster. When an interactive user reclaims a machine, Condor checkpoints the batch job and migrates it to another idle workstation. Condor's ability to manage and schedule jobs on a large number of workstations make it an ideal candidate for inter-job resource manager for parallel applications as well.

The visualization described in this section demonstrates the use of multiple data streams and a complex graphical mapping. The visualization is first created using data from the execution of the application with one distributed scheduling algorithm. The same visualization is then applied to a run of the same application with a different scheduling algorithm, allowing the user to compare the two executions side by side.

Like the ocean model code described in Section 4, this application consists of a master process which generates work steps to be computed by worker processes. Each worker process receives a work step from the master, computes the result, and sends the result back to the master. This paradigm works well in a dynamic parallel programming environment where resources come and go at run-time. When a new resource becomes available, the master can start a worker process there and give it a work step to process. If a resource is lost, the master can give the work step which was being computed there to the next available worker. The application is implemented using the Condor Application Resource Management Interface (CARMI) [10].

The application is a Materials Science program which is designed to predict the properties of new materials based on first principles [13]. Logically, this application consists of two nested loops in which all work for the inner loop (comprising one cycle) must be completed before any work for the next cycle of the outer loop can be started. The inner loop consists of 31 steps per cycle, and the program executes a total of 35 cycles. The steps vary greatly in the length of time they require to compute, but each step varies relatively little across cycles.

The visualization involves two data streams, a work cycle trace generated by the application and a work step trace produced by CARMI. A work cycle record contains cycle duration, occupancy and efficiency information, as well as the number of workers used and number of workers suspended (due to interactive users reclaiming machines) in the cycle. A work step record contains the cycle and step number information, the node number of the workstation that performed the work step, and the timestamps of the beginning and end of the work step.

The number of workstations participating in the computation varied from cycle to cycle, going as high as 23 in some cycles. Since the number was always less than the number of work steps to be performed, some workstations were assigned two or more work steps. All workstations in the cluster had identical physical memory size and CPU speed.

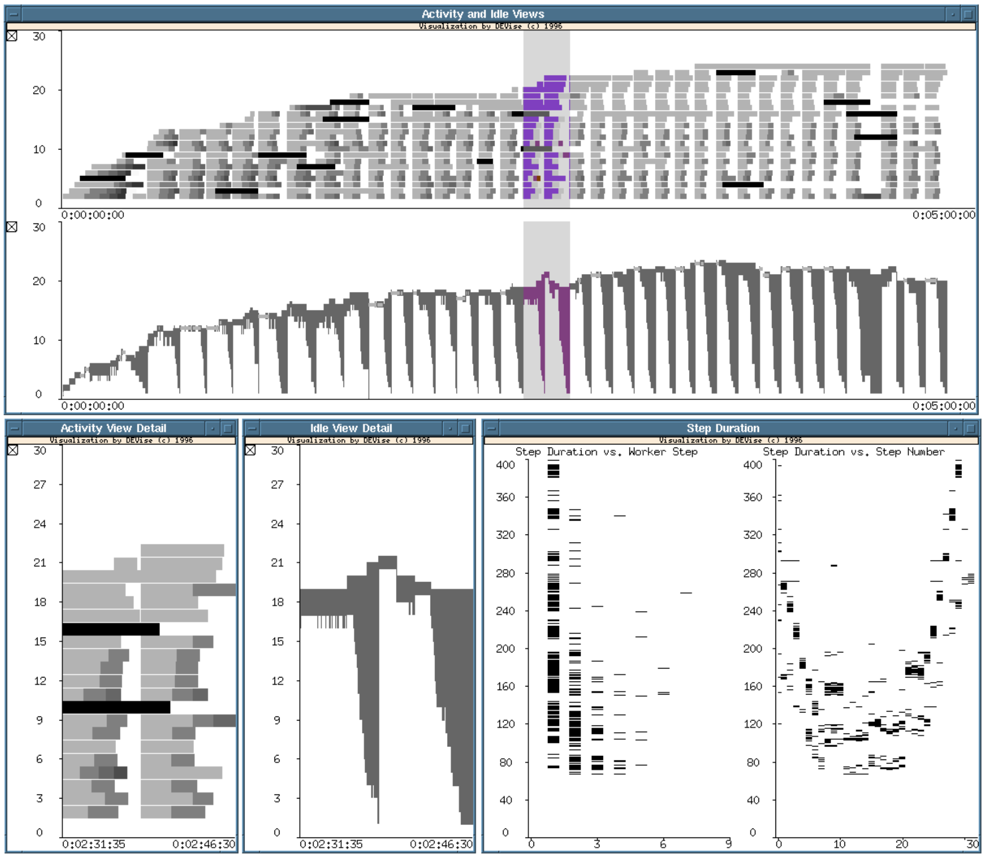

FIG. 6. *Activity and statistical plots of a Condor application. The top two graphs show the activity of the parallel application as a function of time. Two focus views are displayed in the lower left area. The window on the lower right shows statistics on the execution time of work steps.*

The application was first run with a greedy work step distribution algorithm [4]. The resulting work step data stream is displayed in Figure 6. The top window contains two graphs, an *activity plot* and an *idle plot*. Both plots have time on the X axis, measured in days, hours, minutes and seconds. The workstation number is shown on the Y axis. In the activity plot, each colored cell represents a calculation step performed on a workstation. The width of a cell represents the amount of time taken by the work step. The position of a cell on the Y axis indicates the workstation where the calculation took place. The color of a cell shows scheduling information. In each cycle, the first calculation step performed by a particular workstation is assigned a light color, the second step a slightly darker color, and so on. A black color indicates that the calculation was aborted because the workstation was reclaimed.

In the idle plot, the dark areas show the number of idle workstations that are waiting for others to finish computing the remaining steps in the current cycle. White areas indicate the number of busy workstations.

In the center of the activity plot, there is a colored rectangle which highlights a range of GData displayed in that graph. The boundaries of this cursor define another graphical filter. That filter is used in the *Activity View Detail* window where the selected GData are shown as a focus view. From this graph it is easy to tell how many steps a worker executes in a cycle and how long the steps take to execute. A cursor is also set up between the global idle view and the focus view in the *Idle View Detail* window. The user can either scroll or zoom a focus view, which causes the cursor to change location and size, or the cursor can be moved in the global view, which causes the focus view to show the details of another GData subset.

The *Step Duration* window shows additional details of the execution of work steps. The left view plots the duration of steps as a function of the number of steps a workstation performed in a cycle. Each horizontal line segment represents one calculation step. The Y location of the segment corresponds to the duration (in seconds). The X location indicates whether the step was the first, second, third, or higher step a workstation performed in that cycle.

The view on the right shows the duration of steps as a function of step number. The X location of a line segment indicates the step number. The execution times of a work step in different cycles are shown as a vertical stack of line segments. The plot illustrates the large variance in the execution time required by different work steps, whereas less variance exists in the execution time of the same work step across cycles. The variance from cycle to cycle is due to network latencies and slight variances in the workstation load caused by background processes.

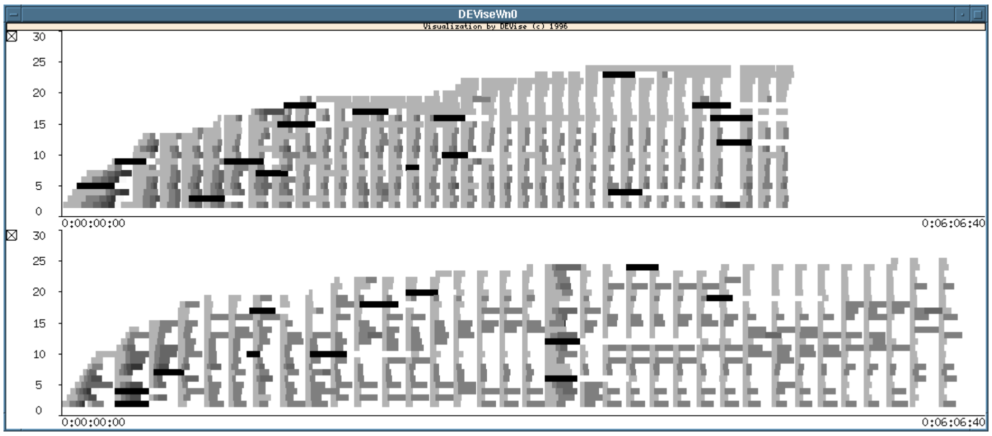

FIG. 7. *Comparison of greedy and sequential distributed scheduling algorithms. The visualization created earlier using one data set was applied to two data sets; the two visualizations are displayed here side by side for comparison.*

Next we compared the greedy work distribution algorithm with the original, sequential ordering of work steps described in [13]. Figure 7 shows the activity plots for both orderings and demonstrates Devise's capability of allowing for easy comparison of related datasets. The visualization we created earlier, in Figure 6, is now applied to two data sets, and the two visualizations can be displayed side by side.

The plots show how the greedy algorithm, in the top view, fares better than the sequential algorithm because it takes less time to complete. We also observe that the greedy algorithm has much less white space between cycles than the sequential algorithm. This indicates that less time is lost due to synchronization between workers at the end of a cycle, so the greedy algorithm is spending more time doing useful computation. Also, we observe that the plot of the sequential algorithm has darker cells, whereas the plot of the greedy algorithm is lighter overall. This difference indicates that with the sequential algorithm, some workstations have to perform more calculation steps per cycle than others. The greedy algorithm provides a more balanced system by scheduling fewer steps on a workstation per cycle.

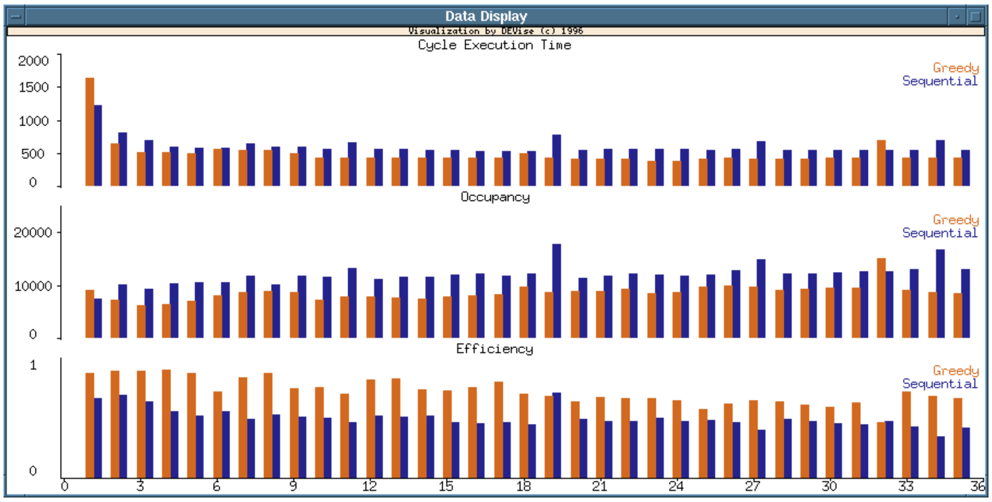

FIG. 8. *Per-cycle performance measures of the two scheduling algorithms. The graphs show how the sequential algorithm consumes more time in all cycles except the first. It also demands a higher occupancy of the resources but uses them with lower efficiency.*

The graphs in Figure 8 compare overall performance measures, including *execution time*, *occupancy*, and *efficiency*. This data is contained in the work cycle trace. Execution time is measured from the time the first step of a cycle is scheduled on a resource until the last step of the cycle has been completed. Condor queueing time is not included. Occupancy is the total amount of resource time allocated to the steps of the cycle, and is viewed as the cost of running an application. Efficiency measures the fraction of the allocated processor time actually used. An efficiency of one means that all allocated processor time was used by the application. Synchronization overhead and network latency are the main sources of efficiency loss.

The top graph in the figure shows execution time as a function of cycle number. The left bar in each bar pair corresponds to the greedy algorithm while the right bar is the sequential algorithm. The greedy algorithm gets a slow start due to a smaller number of available workstations in the first cycle. This is seen in the activity plot in Figure 7. From the second cycle onwards, however, the greedy algorithm consistently completes cycles in less time than the sequential one. The middle plot shows how the sequential algorithm occupies more resources in every cycle except the first. It also uses the resources less efficiently because of the higher number of occupied but idle workstations. This is seen in the bottom plot.

6. Visualization of Condor Cluster Activity. Condor has been running in production mode at the Computer Sciences Department at the University of Wisconsin for the last 7 years. Condor records a multitude of job, host, and cluster information in log files. A separate data stream is recorded for each machine platform in the log. We extracted hourly-sampled data streams of the SunOS/SparcStation and Solaris/SparcStation platforms and created a visualization that compares the activity of these two clusters. The visualization helps a Condor end user to decide to which cluster to submit his or her batch job. The log records contain the following information: the total number of hosts in the cluster, the percentage of hosts used by Condor and by interactive users, the total (relative) load on hosts, and the relative load created by Condor jobs and interactive users.

The top view in Figure 9 shows the total number of hosts in the two clusters as a function of time. The time range of the view is from June 4th to 11th, 1996. The middle view shows the percentage of hosts in interactive use, and the percentage of hosts running Condor jobs is shown in the bottom view. All views are linked on the time axis to each other, so scrolling or zooming in any of the views also scrolls or zooms the other views. The views also demonstrate Devise's capability to overlay two VData sets which allows visual correlation and comparison.

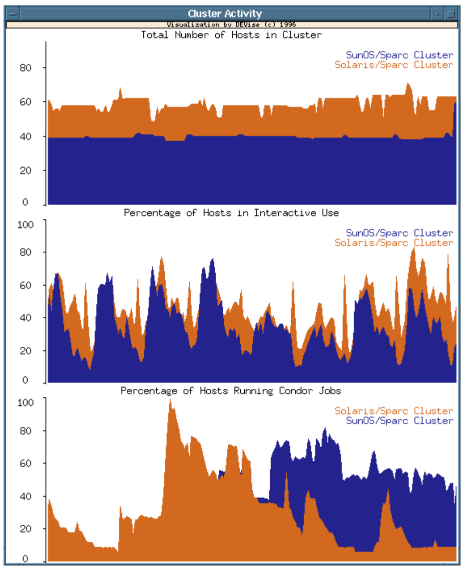

FIG. 9. *Condor cluster activity. The graphs show the breakdown of host utilization among interactive and Condor users. In the middle view, the daily usage pattern by interactive users can be seen clearly.*

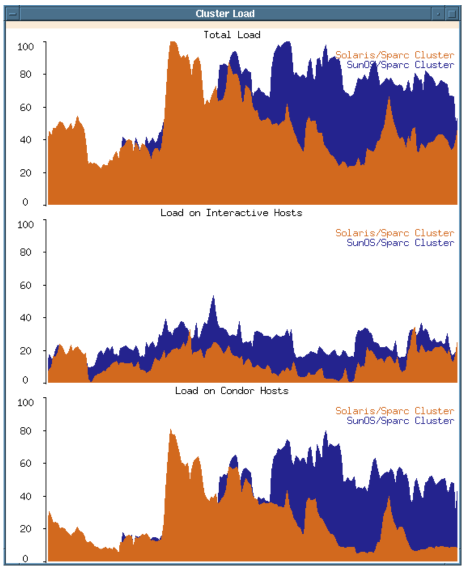

FIG. 10. *Load averages in Condor clusters. The total load on machines has no daily pattern, as Condor keeps machines busy that would otherwise stay idle at night time.*

In the top view we observe that the number of hosts in each cluster remains relatively constant over time and that the number of Solaris/Sparc hosts is consistently higher than SunOS/Sparc hosts. Some variation in the number of hosts is caused by machines taken out of a cluster for maintenance or new machines being added to the cluster. In the middle view we see the daily usage pattern of the workstations by interactive users, with nightly usage being significantly lower than daytime usage. The left-most four peaks in this view correspond to the weekdays Tuesday through Friday and the following two peaks correspond to the weekend days. It is also easy to tell from the view that Solaris machines see more interactive users than the SunOS machines, especially in the evening hours. This is explained by the fact that the machines running SunOS in this cluster were of an older architecture and users preferred to use the newer machines instead.

The top view of Figure 10 shows the total load on machines expressed as a relative number. Each workstation has a load between zero and one (loads above one are treated as one). The sum of the load of all machines is divided by the number of machines in a cluster and shown as a percentage in the view. A value of 100 would therefore mean that all machines were busy. The middle view shows the relative load created by interactive users and the bottom view the load created by Condor jobs.

The lack of any day-to-day pattern in the top view is caused by the load generated by Condor jobs. In fact, that is exactly the reason why Condor exists: to use available CPU cycles when machines would otherwise sit idle (night time). In the middle view we observe that the load on interactive SunOS/Sparc hosts is consistently higher than the load on Solaris/Sparc hosts. This is again explained by the difference in the machine architecture and speed: our SunOS/Sparc hosts are more busy because they are of the older architecture and require more CPU cycles to process user's commands.

We also observe that, for both clusters, the relative load created by Condor jobs is almost identical to the relative number of hosts running Condor (compare bottom view in Figures 9 and 10). This suggests that the load on hosts running a Condor job is at or near one, which is the expected sign that Condor is keeping the machines busy. Intuitively, the difference between the relative Condor load and the relative number of hosts running Condor represents loss of efficiency (wasted CPU time).

The overall message from the graphs of Figures 9 and 10 is that the SunOS/Sparc machines have consistently been under heavier load than the Solaris/Sparc during the period of observation. The higher load is mostly created by Condor jobs, so a Condor user would probably submit his or her new Condor jobs to the Solaris/Sparc cluster.

7. Conclusion. Performance tuning a parallel program involves a number of system components, including the message passing library, performance monitoring tool, resource manager, operating system, and the application itself. Each component produces performance data with a distinct format, frequency and focus. The current practice for visualizing these data streams is to use a separate, customized tool for each data source. In this paper, we have shown how a generic visualization tool can be effectively used for visualizing all such data. Performance analysts benefit from a unified user interface because it lowers the learning curve. An equally significant advantage of integrated visualization is that the system allows side by side comparisons of performance data emanating from different sources. This can lead to better understanding of how the system components interact and work together. Multiple executions of a parallel application with different algorithms, implementations or input parameters can also be compared and analyzed easily.

Being a generic visualization tool, Devise embeds no knowledge of the target domains it is used in and is independent of the type of data used, whether it is from financial markets, clinical medicine, biochemistry, parallel programs or from other domains. Therefore, the key to successful visual integration is to have well-defined data export formats to ensure a "match" between data sets of different sources. Having a configurable data export interface at the source provides the most flexibility. As a case study, we described in this paper how Paradyn was interfaced with Devise to provide more accurate and intuitive visual information about parallel programs. As part of our future work, we plan to extend the interface to allow the user to request additional data from Paradyn based on selections from existing visualizations. This feature would allow dynamic feedback from Devise which might be used to steer the Paradyn tuning session as the application runs.

Acknowledgments. This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant F33615-94-1-1525 (ARPA order no. B550), NSF Grant CDA-9024618, and Department of Energy Grant DE-FG02-93ER25176.

REFERENCES

[1] M. Cheng, *Visual Exploration of Large Amounts of Record Based Sequence Data*, PhD thesis, University of Wisconsin–Madison, Aug. 1995.

[2] M. Cheng, M. Livny, and R. Ramakrishnan, *Visual analysis of stream data*, in Proc. of SPIE – Int. Soc. Opt. Eng., vol. 2410, San Jose, CA, Apr. 1995, pp. 108–119.

[3] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, *PICL: A portable instrumented communication library*, Tech. Report ORNL/TM-11130, Oak Ridge National Laboratory, July 1990.

[4] R. L. Graham, *Bounds on multiprocessing timing anomalies*, Siam Journal of Applied Mathematics, 17 (1969), pp. 416–429.

[5] M. Litzkow, M. Livny, and M. Mutka, *Condor: A hunter of idle workstations*, in Proc. of the Eighth Int. Conf. of Distributed Computing Systems, San Jose, CA, June 1988, pp. 104–111.

[6] M. Livny, R. Ramakrishnan, and J. Myllymaki, *Visual exploration of large data sets*, in Proc. of SPIE – Int. Soc. Opt. Eng., vol. 2657, San Jose, CA, Jan. 1996.

[7] Message Passing Interface Forum, *MPI: A message-passing interface standard*, The International Journal of Supercomputer Applications and High Performance Computing, 8 (1994).

[8] B. P. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, *The Paradyn parallel performance measurement tool*, IEEE Computer, 28 (1995).

[9] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[10] J. Pruyne and M. Livny, *Parallel processing on dynamic resources with CARMI*, in Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph, eds., vol. 949 of Lecture notes in Compter Science, Springer-Verlag, 1995.

[11] R. Ramakrishnan, M. Cheng, M. Livny, and P. Seshadri, *What's next? Sequence queries*, in Proc. Int. Conf. Management of Data, Dec. 1994.

[12] P. Seshadri, M. Livny, and R. Ramakrishnan, *Sequence query processing*, in Proc. ACM SIGMOD, Minneapolis, MN, May 1994, pp. 430–441.

[13] W. A. Shelton, G. M. Stocks, R. G. Jordan, Y. Liu, L. Qui, D. D. Johnson, F. J. P. J. B. Staunton, and B. Ginatempo, *First principles simulation of materials properties*, in Proc. of SHPCC, May 1994, pp. 103–110.

[14] D. Webb, *An ocean model code for array processor computers*, Tech. Report Internal Document No.324, Institute of Oceanographic Sciences, Wormley, U.K., 1993.

[15] P. H. Worley, *A new PICL trace file format*, Tech. Report ORNL/TM-12125, Oak Ridge National Laboratory, Sept. 1992.