Future Directions in Parallel Performance Environments

Allen D. Malony[a]
and Gregory V. Wilson[b]

[a]Department of Computer and Information Science, University of Oregon, Eugene, Oregon 97405, malony@cs.uoregon.edu

[b]Edinburgh Parallel Computing Centre, University of Edinburgh, Edinburgh EH9 3JZ, United Kingdom, gwilson@cs.uoregon.edu

Abstract

The increasing complexity of parallel computing systems has brought about a crisis in parallel performance evaluation and tuning. Although there have been important advances in performance tools in recent years, we believe that future parallel performance environments will move beyond these tools by integrating performance instrumentation with compilers for architecture-independent languages, by formalizing the relationship between performance views and the data they represent, and by automating some aspects of performance interpretation. This paper describes these directions from the perspective of research projects that have been recently undertaken.

1. INTRODUCTION

As the hardware and software technology for parallel systems has advanced, so too has the development and use of performance tools. However, the two have rarely been well-coordinated. Indeed, the increasing complexity of the computational environment brought on by rapid advances in parallel computing technology has created a crisis in parallel performance evaluation. This crisis has forced the development of techniques for parallel performance measurement, analysis, and visualization to counter the growing computational complexity, with the primary goal of increasing *performance observability* [23]. As a result, significant results have been achieved in several important areas: high-resolution timing measurements [22, 34, 35], global clock synchronization [9, 17], hardware performance monitors [2, 19, 25, 32, 31], event-based behavioral abstraction [1, 42], software event tracing [20, 27, 30, 34, 39], perturbation analysis [26], and event-based visualization [4, 12, 24].

One consequence of past work in parallel performance tools is the present concern with what can be termed "performance complexity." The conventional wisdom is that detailed measurements are required to study the time-varying performance behavior of parallel programs [40, 41]. However, there is currently little support for specifying the events which are of interest when investigating a particular performance evaluation problem, or for automatically generating the instrumentation needed to perform the measurement. When measurements can be defined, the volume of data they produce and their potentially complex performance semantics (measured and implied) can quickly saturate the user's capacity to synthesize the data and to perceive important correlations between diverse performance variables. Unfortunately, reducing measurement detail to simplify the interpretation runs the risk of losing critical performance information.

The problem of performance complexity is therefore not one of detail, but rather one of understanding. Parallel system users cannot be expected to understand all of the issues related to performance, and, therefore, are and will be ill-equipped to deal with many aspects of the parallel performance evaluation process, including problem specification, tool application, data interpretation, and performance optimization.

Recent trends in parallel processing suggest that the issues of performance observability and complexity will become more difficult. Massive parallelism has been adopted as the salvation for computationally intensive applications. The notion of scaling the problem size has been proposed as a way to achieve performance scalability on machines with a large number of processors. The shift from control parallelism to data parallelism as a programming paradigm – for reasons of programmability and performance scalability – is evidenced in recent parallel Fortran [6, 8], C [11], and C++ [21] language implementations. At the same time, parallel execution models that abstract the underlying parallel machine to hide machine-dependent details of their implementation from the user are becoming more prevalent (e.g., the Single-Program Multiple-Data (SPMD) model or the Linda tuple space model). These trends – increased performance through increased parallelism and scalability, plus increased programmability through abstraction and machine independence – pose a new challenge for performance tools. Indeed, the "grand challenge" problem in computational software for scalable, parallel systems is the development of tools that can routinely (and, ideally, automatically) produce high-performance applications.

In addition to the advances in performance tool technology that must occur to provide users with a higher-level interface for measuring, analyzing, visualizing, and interpreting parallel performance data, we advocate the notion of a fully-integrated toolset (a *performance environment*) embedded in and using the resources of a parallel programming system. The idea is to provide an environment for solving performance problems which relieves the user of much of the manual effort of performance observation while reducing the intellectual burden of evaluating complex performance behavior. Clearly, such an environment cannot spring into being overnight, especially if it is to leverage parallel programming technology (e.g., program analysis tools and compilers). It is therefore important to concentrate at present on the needs and requirements for tool integration, the design of techniques that promote performance and program tool interaction, and the opportunities for environment development.

In this paper, we study three directions that will be interesting to pursue for future parallel performance environments. In Section 2, we consider the problem of measuring parallel programs based on the SPMD execution model. The principal issue here is performance observability, and in particular, how performance measurements can be specified and implemented in a SPMD parallel programming environment. Parallel performance visualization is the topic of Section 3. Here we propose a performance visualization model which emphasizes specifying performance and view abstractions and mapping these to one another. Actual visualization generation is postponed until this formal specification

is defined. In contrast to current performance visualization tools, this allows complex performance views to be imagined and rendered through a variety of graphical means. Finally, in Section 4 we introduce the concept of a performance expert system. The aim here is to provide the user with a tool that acts as an intelligent performance assistant, automating measurement and analysis tasks, providing critiques based on known performance models, directing performance optimization efforts, and predicting the performance impact of program or system changes.

2. Performance Measurement of SPMD Computations

The Single-Program Multiple-Data (SPMD) parallel execution model has been advocated as a basis for languages for massively parallel systems because of its scalability, programmability, and portability. In SPMD programs, all processors execute the same program, but take advantage of memory locality by operating primarily on local data. The parallel data structures in the program are distributed across the processors in a fashion that enhances parallelism. When local computation involves data from other processors, some form of communication is invoked to retrieve the data. Because algorithm designers often think in terms of synchronous operations on distributed data structures [15], SPMD programming languages allow massively parallel computation to be easily specified. Furthermore, because the SPMD execution model is an abstraction of a machine's low-level operation, the user is typically presented with a hierarchy of concurrent programming abstractions based on SPMD semantics that insulate her from machine specific implementation issues, thereby increasing the portability of both the language environment and resulting parallel programs.

For example, Fortran D [6, 8] and pC++ [21] are two parallel programming languages which assume an SPMD style of execution. Fortran D is an enhanced version of Fortran that provides data decomposition specifications for *problem mapping* (to maximize memory locality and reduce data movement given an unlimited number of processors) and *machine mapping* (to translate the problem onto the finite resources of the machine). These two levels of parallelism are abstracted by explicit decomposition, alignment, and distribution statements in the language. This implies that selecting a data decomposition strategy is the task of the user. The Fortran D compiler translates a Fortran D program into SPMD code with explicit message communication for execution on MIMD parallel machines. The compiler partitions the program using the *owner computes* rule whereby each processor is responsible for computing the value of data it owns [3].

Parallel C++ (pC++) is a language based on an object-oriented, parallel programming paradigm called the *Distributed Collection Model*. In the model, programmers can keep a global view of a whole data structure as an abstract data type, while specifying how the global structures are arranged and distributed among differ processors to exploit memory locality. Parallelism in pC++ programs is defined by the parallel action of class operators on the individual elements of a large collection of related objects. Users control the distribution of objects of a collection to processors and memory hierarchies. This "object parallel" concept subsumes all the features of "data parallel" computation with the ideas in Fortran D data distributions, but also incorporates the techniques of abstraction and specialization that make object oriented design powerful.

The advantages of using high-level languages for parallel programming based on the SPMD model – concise coding, parallelism abstracted through distributed data structures, and portability – is, seemingly, a disadvantage when programmers try to tune applications for a particular system. Because the language presents an execution model to the user which hides the underlying parallel system environment, it restricts user-level access to performance data. Such access can be critical to evaluating a program's execution or comparing alternative data decomposition strategies (e.g., message communication overheads, levels of parallelism, timings of pe- processor computations). Clearly, what is required is an SPMD measurement strategy that is integrated with the SPMD language compiler.

2.1. Measurement Strategy

There are two important performance observability issues that arise when deciding on an SPMD measurement strategy. The first concerns how performance measurements are specified and at what level. In particular, there is the question of whether performance instrumentation should be tied to SPMD parallel program abstractions. Although languages such as Fortran D and pC++ prohibit direct measurement (i.e., from the user program) of the performance of the underlying parallel system, the programming abstractions they provide to the user are central to the interpretation of any performance data collected. Thus, it is reasonable to think that performance instrumentation should be specified at the language level and should be based on program semantics. In this sense, one could provide performance instrumentation as first class objects or statements in the SPMD language, able to reference other language objects, assign instrumentation attributes, and possibly provide a mechanism for the program to read performance data during computation. For example, for the pC++ environment, we are developing a system of program instrumentation annotations that can be used to select measurement alternatives.

The second issue concerns how performance measurements are implemented. Clearly, performance instrumentation specified at the language level cannot by itself describe what low-level data are needed to obtain the desired performance measurement, nor how those data are to be collected. Rather, instrumentation specifications at this level must be processed in association with the SPMD language compiler if performance measurements are to be properly realized. For instance, the instrumentation may specify that an event should be captured every time a particular section of a data structure is accessed. In this case, the performance measurement generated must take into account how the SPMD language compiler generates code to partition, distribute, and access the data structure. In general, when program transformations are performed by the compiler, it is necessary for the performance tool interpreting the instrumentation specification to adapt the measurement support to ensure consistency and efficiency.

The contrast implicit in the above is one between performance measurement *policy* (i.e., how performance instrumentation is specified) and performance measurement *mechanism* (i.e., how the performance measurement is implemented). The high-level nature of SPMD parallel languages (and of other parallel languages that abstract low-level details of the execution environment) forces measurement policy and mechanism to be separated if the parallel performance environment is to fully support the performance evaluation needs of the user. It is unreasonable to expect a single solution combining measurement policy and

mechanism (for instance, instrumented communications libraries such PICL [7]) to fulfil these requirements. However, separating policy from mechanism requires the performance tools to be integrated with other tools in parallel programming environment; in particular, with program analysis and compiler tools.

2.2. SPMD Measurement Examples

Because SPMD parallel languages present an abstraction of the execution environment to the user, they can be targeted to different machines and architectures. It is important that performance instrumentation (i.e., measurement policy) be consistent to enhance instrumentation portability and allow cross-machine performance comparisons. Obviously, the underlying performance measurement implementation will change, depending on machine characteristics. To simplify our discussion, we will consider a general MIMD distributed memory machine in the following examples. The examples are not complex, but even so they demonstrate the need for, and the potential capabilities of, new SPMD performance measurement support.

The first example comes from a paper on compiling Fortran D for MIMD distributed memory machines [13]. Consider the following code segment:

```
REAL X(N,N)
do i = 2, N
   do j = 1, N
      X(i,j) = X(i-1,j)
   enddo
enddo
```

Fortran D provides data decomposition specifications that allow the matrix X to be decomposed across processors in the machine and operated on in a distributed manner. Figure 1 shows three possible decompositions of X (row blocked, column blocked, row and column blocked) and the cross-processor dependencies these lead to.

Suppose we want to capture a trace event each time a row of X is updated and each time a remote communication takes place in any processor. To satisfy the first measurement requirement, one might think of placing measurement code between the i and j loops. However, using source-level statements, it will be difficult to express the semantics of the instrumentation with respect to the SPMD parallelization which the compiler might perform. For example, if a row is distributed across processors, how does the user express in the language what is meant by "at every point when a row of X is updated"? Furthermore, source instrumentation may actually restrict the set of parallel transformations possible. Satisfying the second measurement requirement is even more difficult, since message communication operations are hidden at the source level.

As referred to above for the pC++ environment, providing a set of annotations for specifying instrumentation semantics is a reasonable approach to providing a measurement policy. Annotations appear merely as comments in the source, but allow the program objects and semantics to be used to express instrumentation requirements. Clearly, such annotations would have to be interpreted by a performance tool responsible for implementing the measurement and, thus, must adhere to some defined language. The important point is that the annotation language (and the performance instrumentation that it specifies) is not constrained by SPMD language syntax.

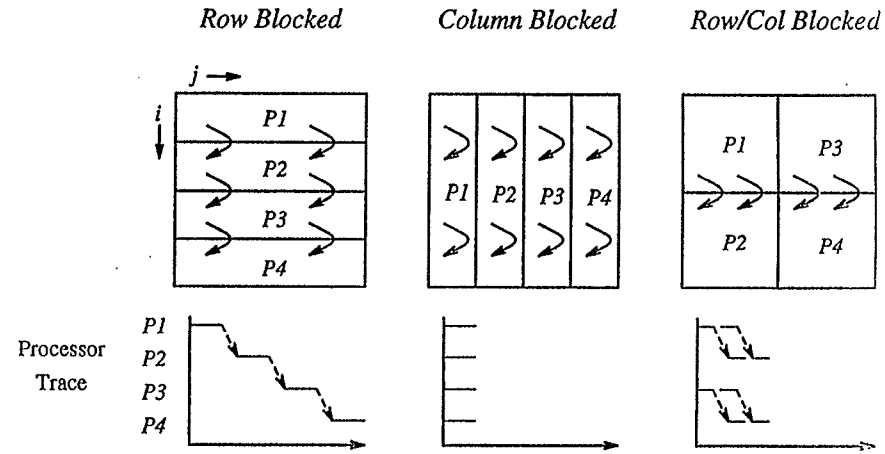

Figure 1. SPMD Decomposition, Execution Dependencies, and Performance

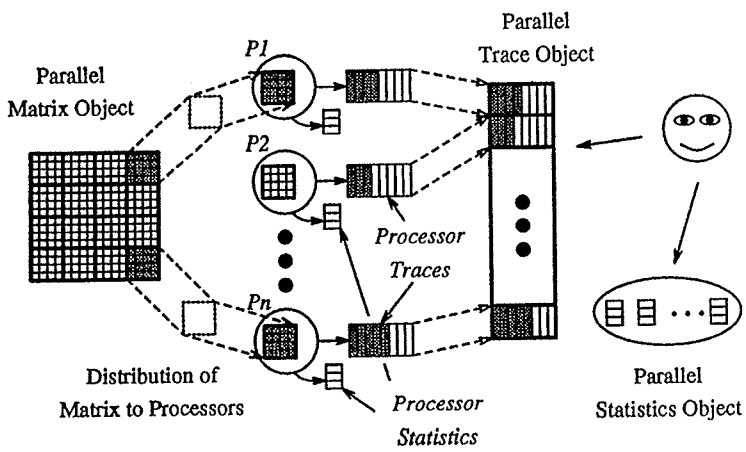

Figure 2. SPMD Performance Measurement

In the example above, we might use an annotation like:

```
C$ann  [ IfDef(INSTRUMENT)
C$cont   for this processor{
C$cont     when X(i,j) updated for each local i and for all j
C$cont     RecordEvent(X_ARRAY_UPDATE, i)
C$cont   }
C$cont   RecordMessageComm()
C$cont ]
```

Here the semantics of the instrumentation is clear: a separate event is recorded (in each processor trace) when the entire row segment of X within that processor has been updated, and whenever a message communication operation takes place. The instrumentation specification is interpreted by the performance tool implementng the measurement for each program execution case, thereby binding the instrumentation semantics to the program parallelization. In Figure 1, we show execution time graphs that might result from such measurements, highlighting the message communication events. The key point is that the instrumentation specification (i.e., annotation) remains the same.

Our second example is shown in Figure 2. Here, a matrix is distributed across processors. We specify two types of measurement: recording events corresponding to selective matrix updates, and accumulating profile statistics (counts and times) of selective parts of the program involving matrix computations. As in the previous example, each processor performs the specified measurements with respect to its own local data (i.e., instrumentation activation can be conditioned on the attributes of processor-local data), creating a local trace buffer of events and an array of profile statistics. Our interest in this case is not what the measurements are, but how performance data might be accessed.

One interesting extension of the annotation approach is the idea of using annotations to give a program access to the performance data collected. The ability to access runtime performance information is important for supporting functions such as real-time parallel algorithm animation, computational steering, and adaptive parallel execution. Suppose, for instance, that within an instrumented block of matrix computations, the programmer wanted to know the number of times a particular computation was performed or how long a set of computations took. She might use an instrumentation annotation to assign a program variable to those performance values[1]:

```
C$ann  [ IfDef(INSTRUMENT)
C$cont   count = GetCount(COMPUTATION_A)
C$cont   time = GetTime(COMPUTATIONS_ABC)
C$cont ]
C$ann  [ IfNotDef(INSTRUMENT)
C$cont   count = -1
C$cont   time = 0
C$cont ]
```

Here we see how objects of the performance measurement can be made, in a sense, first class objects of the SPMD parallel language. It is also important to note that the performance measurement itself can leverage the semantics of the SPMD language for its implementation. For instance, in Figure 2 we have shown two high-level, data parallel performance objects that give an SPMD view of the performance data. Thus, operations that the performance measurement system might need to perform on its own behalf (such as combining all processor traces into a single time-sequenced trace at the end of program execution) or on behalf of the user (such as determining the average amount of time spent by any processor in a section of code for the last iteration) can be implemented using the features of the SPMD language.

In summary, as parallel languages and environments continue to raise the level of the programming interface, thereby abstracting away execution characteristics of the underlying parallel system, performance measurement tools must adjust by separating instrumentation specification (i.e., measurement policy) from measurement implementation (i.e., measurement mechanism). Although this will require a tighter integration of these tools with other tools in the programming environment, it is encouraging to note that the high-level execution semantics of SPMD parallel languages might provide a useful metaphor for performance measurement abstraction.

[1]Care must be taken by the instrumentation preprocessor or by the programmer (as is the case here) that the program variable used has a default value when the instrumentation is disabled.

3. PARALLEL PERFORMANCE VISUALIZATION

> What is to be sought in designs for the display of quantitative information is the clear portrayal of complexity. Not the complication of the simple; rather the task of the designer is to give visual access to the subtle and the difficult — that is, the revelation of the complex.
>
> Graphics *reveal* data.
>
> — Edward Tufte, *The Visual Display of Quantitative Information*

Performance visualization is the use of graphical display techniques for the analysis of performance data in order to improve understanding of complex performance phenomena. Although significant advances have been made in visualizing scientific data [10, 5], techniques for visualizing performance of parallel systems remain *ad hoc*. Unfortunately, the need for better visualizations of parallel performance will become more acute as parallel systems become more complex. A more robust performance visualization paradigm is required if we are to use sophisticated computer graphics techniques to present complex performance data.

Tufte's statement above implies that one useful approach to performance visualization might be to begin with an understanding of the complexities of the performance data (i.e., what the data represents) and then consider how graphics could be used to "reveal" important aspects of the data. Unlike scientific visualization, the graphical display of information about the performance of parallel systems rarely has any real physical basis; rather, the data represent relationships and interactions between logical or virtual objects. By implication, renderings tend to be more abstract and impressionistic, constrained less by "reality" and based more on parameters controlling the graphic design space. This lack of a direct mapping complicates performance visualization. Furthermore, the most useful sets of visualizations are likely to be determined only from experience, and will depend on the needs and personal preferences of the user. These arguments suggest that performance visualization research could benefit greatly from an integration of performance evaluation, application domain expertise, human-computer interaction (HCI), visual perception, and graphic design.

Although significant intellectual effort has been devoted to the design and development of performance visualization tools [4, 12, 27, 24], much of the software is specialized and experimental, and cannot easily be extended to provide new functionality or to be applied to new performance evaluation scenarios. In part, the problem is due to the custom nature of the tools developed (e.g., only accepting performance data of a specific type, restricting analysis and display options, or incorporating non-reusable user-interface and graphics software). Recent work [33, 16] has addressed the issues of reusability and configurability by providing toolkit-based technology that allows analysis and display modules to be composed into a performance data visualization graph. To promote inter-operability and flexible configuration, the analysis and display modules in these environments are generally based on simple functional transformations; standard mechanisms are then provided for module composition and interaction. The generality of the analysis and display modules requires that their implementation be independent of performance semantics. However, for the visualization to be meaningful, performance semantics must be expressed. The

configuration of modules is therefore an encoding of performance semantics through a mapping of analysis module output to the graphic display parameters. Because of this, the environment often does little more than provide a visual interface for module configuration to aid in the semantic binding between analysis and display – much of the burden is placed on the user for constructing semantically meaningful visualizations. For this reason, it may be difficult to incorporate sophisticated graphics in existing performance visualization environments in other than naive ways.

If performance visualization is to become an integral tool in parallel performance evaluation, it must be based on a formal foundation that relates abstract performance behavior to visual representations (i.e., "visual performance abstractions"). It must be possible to formally represent performance visualizations as functional *mappings* of semantic *performance abstractions* to *performance views*, independent of graphics display technology. These representations can then be used to produce performance visualization software by generating interfaces to graphical programming libraries or input data to existing visualization systems[2]. In this manner, new parallel performance visualization techniques can be quickly developed and studied. In addition, techniques from HCI and visual perception can be more readily applied and tested.

3.1. PARASEER: A Parallel Performance Visualization Project

We discuss this future direction in parallel performance visualization against the backdrop of the PARASEER project just beginning at the University of Oregon. PARASEER is an exploratory research study of next-generation parallel performance visualization technology [28]. The objectives of the project are to:

- Define a formal framework for describing new parallel performance visualization techniques based on the paradigm of mapping performance data abstractions to performance views.
- Develop techniques for generating parallel performance visualization software from abstraction, view, and mapping specifications and displaying the results with object-oriented display extensions to graphic user-interface software and existing data visualization systems.
- Construct new parallel performance visualizations and evaluate their usefulness in real parallel applications.
- Apply HCI and visual perception technology to performance visualization.

Figure 3 shows the model of parallel performance visualization on which PARASEER will be based.

The first step in visualizing performance is to define abstractions that embody performance semantics. This is done by specifying input parameters, performance "state," and execution rules or procedures in abstraction definition. In this way, we can think of a *performance abstraction language* being used to build a "performance object" whose definition encapsulates its semantics and behavior. The language further supports the composition of objects for the purpose of higher-level performance abstractions. In PARASEER,

[2]The nascent work in this direction [38, 18, 37] highlights the potential power of this approach, although the prototype tools are still limited by their interface to powerful data analysis and graphics software.

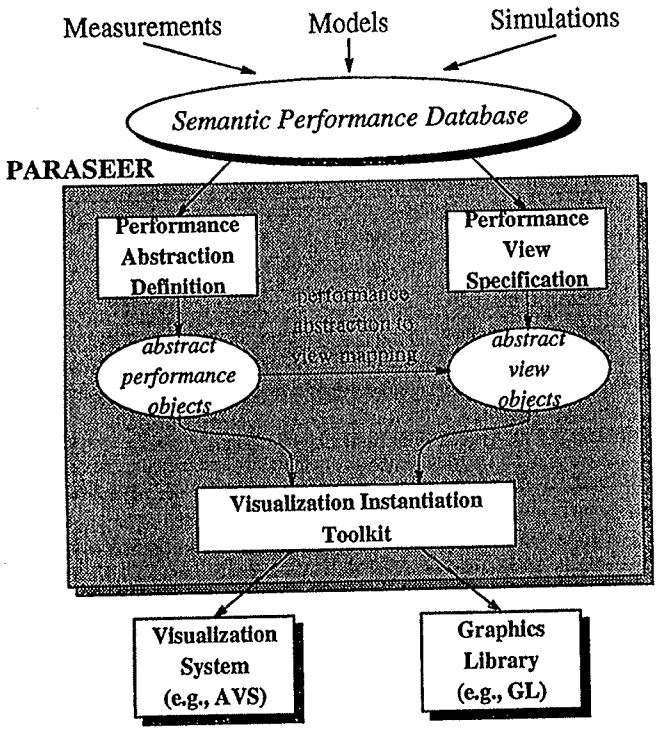

Figure 3. The PARASEER Parallel Performance Visualization System

we intend to develop a performance abstraction language combining the prior work of Snodgrass [42] and Ogle [14] with recent developments in object-oriented languages.

Like performance abstractions, performance views encapsulate the appearance and behavior of graphic displays. Again, a formal language, the *performance view language*, is used to construct "view objects." Of particular importance to view object definitions are the algorithms that describe various geometric properties and transformations of views in a virtual coordinate space as a function of view object inputs. The performance view language should be object-oriented, supporting the structural composition of primitive performance views. The descriptions of graphic appearance can be supported by functional, geometrical syntax in the language.

The binding of a performance abstraction to a performance view – mapping performance object outputs to view object inputs – conceptually generates a performance visualization. As we can develop languages for performance abstraction and view definition, so too can the mapping be formally specified. The *mapping language* might use the notion of type matching between abstraction output and view input parameters to determine compatible combinations. It is also reasonable explore the use of coercion to match abstractions with views. Finally, procedural semantics can be incorporated in the language to specify abstraction and view interaction (e.g., view animation via changes in performance object state).

3.1.1. Some Illustrative Examples

As a simple example of these ideas, consider the performance abstraction of load balance among a set of processors mapped to two views of processors: one as a vibrating string in two-dimensional space and the other as balls moving in a three-dimensional space. The load balance performance abstraction is a composite object containing P processor utilization abstractions, where P is the number of processors in the system. The processor utilization abstraction contains the percentage utilization of a processor over a specific time interval. Let us assume the utilization information is broken into *user*, *system*, and *idle* time percentages. The load balance abstraction accepts a vector of numbers representing processor utilization data and specifies how this data is combined to produce utilization statistics concerning global load balance (e.g., average processor utilization and variance). As a result, new data might be passed back to the processor utilization abstractions to indicate the amount of processor deviation from the global average.

Given the load balance abstraction, let's consider using the following view abstractions for its visualization. The first is a vibrating string view. We think of each processor as representing a point along a string of equally-spaced points; this abstraction would be most appropriate when the number of points is anticipated to be large (e.g., $P = 1000$). The "shape" of the string is determined by how much each point deviates (plus or minus) from an average value – if each point has zero deviation, a straight line results. Clearly, we can map the load balance abstraction to this view abstraction by assigning the string's average value to the average utilization (e.g., user utilization) and each string point deviation to the processor deviation. In this manner, as the performance abstraction processes new processor utilization input, the view abstraction is "animated", and the string takes on a new shape (Figure 4). At this point, only the graphical generation of the visualization remains. It is interesting to note, though, that in this case certain geometrical and

Figure 4. Vibrating String View Abstraction for Load Balance

presentation attributes of the visualization (e.g., size, orientation, string color) are left unspecified because they are not important in the view abstraction.

A more complex view abstraction for load balancing might describe balls in a three-dimensional space where ball position and motion are targets for mapping the load balance abstraction. Suppose we think of the three axes of the space representing the three utilization values: user, system, and idle. A ball's position, therefore, in the space is determined by these coordinates. Visually, we will perceive load balance as spatial relationships between the balls. A unique ball could be used to show the global utilization average. Averaging the utilization values of each processor using a sliding time window whose duration was defined by the user would animate the display. Shortening the window would make the balls more "responsive" to small changes in utilization, but the resulting "jitter" could make the display difficult to interpret. Lengthening the window, on the other hand, would give a more settled, but possibly less informative, display.

A last example of a complex view that we hope to generate automatically with PARASEER is the "kaleidoscope" defined for a fine-grain concurrent logic programming language [43]. The basic concept is simple: we display the dynamic call graph (process tree) in polar coordinates so that computationally intensive nodes are afforded proportionally more space. Another performance dimension can be represented by color; for example, in [43] nodes were colored according to time of function invocation. Creating views of this complexity requires algorithmic specification (as opposed to pure functional composition, as in toolkit approaches), if for no other reason than ease of programming. The PARASEER notion of performance abstraction and view specification languages will allow rapid exploration of alternative heuristics for coloration and node placement. It is also important to look at the configuration effort to apply a new view, such as the kaleidoscope view, to another performance abstraction, for instance load balancing. Again, we hope to significantly reduce the effort with a formal abstraction framework and binding language.

3.1.2. Instantiating Views

The ultimate goal of the PARASEER project is to explore new parallel performance visualization techniques and to evaluate their effectiveness in real parallel performance problem domains. In addition to the formal framework for performance visualization design, PARASEER will take two approaches to visualization instatiation. The first will build interfaces to existing data visualization software packages (e.g., AVS, used extensively for scientific visualization) to provide a flexible environment for graphics prototyping and control. This approach leverages these tools' capabilities for handling large data

sets, applying sophisticated graphics functions, supporting distributed processing, and extending the visualization architecture.

The second approach will develop a performance display library as an extension to graphical user interface software to provide more programmatic performance visualization support. The concern here is that AVS and other visualization packages might not provide enough flexibility for certain visualization problems. We want to be able to develop executable "targets" (using, for instance, Silicon Graphics' graphics language GL or their Iris Inventor 3-D object library) for visualization instantiations at the program level, rather than the application level. The parallel performance visualization will still be formally specified, but the user will able to construct custom performance visualization tools programmatically, with graphic display objects closely associated to the view specification. The second approach is necessary to build performance visualizations that operate in real time.

3.1.3. PARASEER Validation

We intend to validate the PARASEER approach by defining performance abstractions and views for commonly found parallel performance visualizations in tools such as Paragraph [12], Pablo [33], and TraceView [24]. The goal is to evaluate the generality and flexibility of the PARASEER framework by formally specifying the visualization types found in these tools. In this work, we will use the two approaches discussed above to instantiate visualization specifications. If the PARASEER method cannot easily specify existing visualization types and show that there is a path to instantiate the visualization, it will be difficult to justify its use in more sophisticated visualizations.

3.2. The Role of Computer Graphics in Performance Visualization

Most parallel performance visualization techniques that have been developed for use with today's high performance graphics workstations are inherited from the graphic arts field. Many images are simple two-dimensional representations evolved from early displays of statistical data. This is quite natural because until graphic workstations became available, artists and graphic designers working with traditional media were the only way that technical illustration could be done.

There are, however, numerous computer graphic techniques which improve upon and go beyond the traditional artistic methods. For example, in the area of color, it is possible with today's workstations to compute color scales from perceptually uniform color spaces and use these scales to display the variation of a parameter related to parallel performance [29]. Transparency techniques can be employed to place a colored filter over an object and relate the density of the filter to the value of some variable [36]. Other computer graphic image synthesis methods remain to be exploited in parallel performance visualization. Correlating surface roughness or surface texture with a parameter that tracks a performance variable or using shadows as a visual cue for the existence of performance phenomena has obvious appeal. Shading algorithms can also be adapted to allow parallel performance parameters control diffuse and specular reflection. Given the discrete form that the data for visualization problems typically assumes, volumetric rendering techniques are easily adapted to the problem and should be explored more fully.

Together, these techniques open up exciting possibilities for displaying subtle differences and complex interactions among performance variables. In addition, the huge volume of

information generated in high-performance, massively parallel systems will require the use of high-bandwidth graphical images. The PARASEER project represents an effort to develop a formal infrastructure for advanced parallel performance visualization research. Its purpose is to significantly reduce the software implementation burden often encountered in performance visualization projects, freeing the researcher to concentrate on the fundamental properties of the performance abstractions and views that together represent a performance visualization.

4. Using Expert Systems in Parallel Performance Evaluation

The final future direction we will discuss is also the most speculative. As mentioned earlier, programming parallel computers in a language which abstracts away the characteristics of particular hardware platforms can be a disadvantage when programmers have to tune applications for particular systems. While it may be desirable, when designing an algorithm, to ignore details such as the number of processors, their interconnection, or even whether they operate synchronously (SIMD) or asynchronously (MIMD), it is exactly these attributes which are important to performance optimization.

A partial solution to this problem is to allow programmers to annotate their programs (using "pragmas" or compiler directives) in order to express machine-dependent optimizations. However, the combination of the number of attributes of the underlying system which such annotations might reflect and the different ways programming constructs could be implemented would lead to a bewildering variety of choices. A more attractive solution in the long term is to automate, or partially automate, performance evaluation and tuning. In this section we describe our plans for a performance expert system that combines advances in tools for parallel performance analysis with AI technology for knowledge-based analysis to function as an "expert performance advisor". The prototype system we are now developing is named PERCEPT (PERformanCe ExPerT), and is intended to demonstrate how expert systems technology could be used in parallel program evaluation and optimization.

4.1. Automating the Performance Optimization Process

Optimizing a program's performance is an inherently cyclic process, in which code development, instrumentation, execution, and evaluation are done repeatedly. While today's state-of-the-art performance analysis systems offer little more than basic measurement and analysis facilities, the ideal system would serve as a "performance advisor", using observed performance characteristics and stored knowledge to direct optimization efforts. We believe that the development of more measurement and analysis tools which leave the intellectual burden of performance interpretation on the application developer will, by itself, do little to improve the average programmer's productivity. Instead, such tools must be integrated with advances in performance modeling and prediction to create a system capable of automatic, "intelligent" performance optimization.

Such a "smart" performance optimization system would couple an intimate knowledge of the semantics of parallel computations with models (measured and theoretical) of the overall nature of parallel performance behavior at the program, system, and machine levels. It would include:

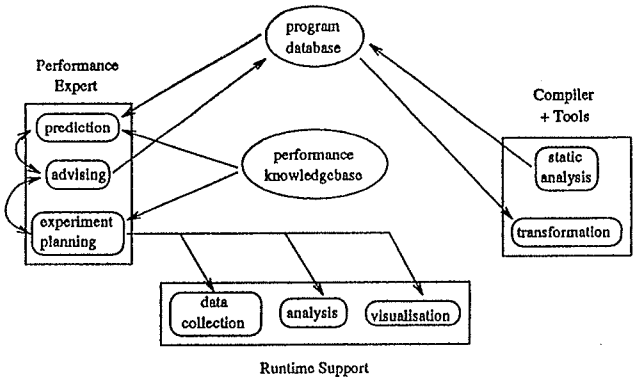

Figure 5. System Model of an Expert System for Performance Tuning

- an extensive performance measurement framework for capturing data at all levels;
- a set of tools for automatically and interactively characterizing observed performance phenomena and building a database of performance knowledge;
- techniques for abstracting observed performance data to create models of performance behavior; and
- an approach to performance prediction that would rely on performance models and known relationships between computational properties and performance characteristics to forecast the effect of changes in the program or execution environment.

The difficulties in building such a system are as daunting as the problems the system is trying to solve. The level of sophistication such system would represent has never been achieved, and will require advances in the states of several arts. However, even partial failure will be informative, as it will help clarify which aspects of performance tuning may be considered "routine", or domain-independent, and which can only be done using domain knowledge.

4.2. System Goals

The basic model of PERCEPT is that the system should help the user design and execute performance evaluation plans. For a plan to be realized, it must:

- specify the parallel program for evaluation;
- specify the target environment for the program's execution;
- identify performance objectives to be achieved;

- identify critical performance factors related to the objectives;
- define a set of experiments to measure the performance factors;
- specify a measurement approach for each experiment;
- select analysis alternatives to be applied; and
- request performance results to be generated.

On one level, PERCEPT will be just a set of tools for measurement, analysis, and results presentation. The user will choose different combinations of tools to apply by actuating different parts of a plan. At a more sophisticated level, PERCEPT will help the user construct and execute performance evaluation plans according to strategies that it has been given. These strategies will embody knowledge about how the different components of a plan are interrelated, about good and bad plans for different performance objectives, and about how performance results relate to known problems.

Deciding what performance measurements to make, the set of experiments to conduct, and the type of analyses to perform to produce results that address a particular performance question is (to say the least) non-trivial. However, we believe that strategies can be specified for a variety of generic parallel performance problems, particularly those related to data decomposition and load balancing. One of the aims of this work is to determine just how much of program tuning is generic in this sense, and how much relies on domain-specific knowledge.

As an example of a generic tuning problem, suppose we want to determine load balance during a particular phase of a parallel program's execution. A strategy for doing this would describe instrumentation to detect changes in processor state, measurements that capture processor utilization data at different time scales (i.e., traces versus samples), analyses to produce statistical profiles of load balance at different perspectives (local and global), and criteria for evaluating the quality of load balance performance. Such a strategy could then be incorporated into a hierarchy of more complex strategies, such as strategies for observing the effects on load balance as the number of processors vary.

A more complicated problem area in which domain knowledge might play an important rôle is choosing a combination of data decomposition and work allocation which together give highest performance in a particular program. For example, consider the simple dynamic programming code discussed in [44]. The main data structure being manipulated is an upper-triangular matrix; in order to update a point (x, y), all points between it and the main diagonal in its row and column must be accessed. On a distributed-memory machine, a simple row-wise or column-wise decomposition of the matrix would allow half of all accesses to be local, but would force the other half to be non-local. Duplicating the matrix, and storing one copy by row and the other by column, can reduce these memory access costs; caching previously-fetched rows and columns, and allocating work to processors so that cached values can be re-used as often as possible, affords a similar work reduction. There is clearly some scope here for a tool to recognize data access patterns (using information provided by the compilation tools and observation of actual program performance), to help design experiments to compare alternative caching and

work allocation strategies, and to record, analyse, and report on the results of these experiments.

A separate problem from the notion of designing strategies for performance analysis is the concept of performance problem recognition. The most common question of parallel systems users is, "Why is my program performing as poorly as it is?" rather than simply "What are the performance characteristics of my program?" The approach which we will adopt in PERCEPT will be based on model abstraction and identification. We will attempt to create abstract representations of a program's execution, annotate these with actual performance data, and match them against archetypal performance models. Execution abstractions will be derived from:

- user supplied execution specifications (e.g., flow graphs);
- control and data flow information from static program analysis; and
- pattern analysis of performance data (e.g., computational phases).

Once an execution abstraction has been formulated, it will define a basis for experimentation within which different parameters of the execution model can be evaluated. These experiments will generate a performance database which will be used in the recognition process.

4.3. Evaluating the Expert System

While the other tools discussed in this paper represents extensions of existing research directions, PERCEPT is a step into the unknown. It will therefore be as important to evaluate its failures as its successes, in order to guide further research. The most important evaluation criterion that will be used is also the simplest: can programmers improve the performance of their programs more quickly using the tool? If the answer is "no", then it will be important to determine why not — is the overhead of using PERCEPT outweighs the performance gains it realizes, is most performance optimization done using domain-specific knowledge which PERCEPT does not, or cannot, incorporate, etc. While a positive answer to the first of these would call the whole approach into question, the second would indicate that the direction for follow-on research should be toward developing knowledge capture tools capable of encoding what experts know about particular domains.

A second evaluation which will be made of PERCEPT is the degree to which it can detect key aspects of program performance, and present these to users in a comprehensible fashion. Once a programmer knows why a program is performing poorly, it is often relatively straightforward for her to alter its behavior accordingly; the hard part is determining what the bottlenecks actually are. The main challenge we foresee will be to find a representation which will make model classification and matching possible. One option which will be explored is to have the programmer implementing a library encode her understanding of how it is likely to behave, and what aspects of its behavior are likely to be critical to performance on particular architectures. This knowledge would then be available for PERCEPT to use directly in designing and analyzing experiments, and, through PERCEPT, to the programmer using the library, so that she could avoid performance problems rather than correct them after the fact.

5. CONCLUSION

Each of the advances outlined in the previous sections is ambitious; their combination will undoubtedly be difficult to achieve. However, as the complexity parallel computing systems continues to grow – as more complicated memory hierarchies and caching strategies are introduced, for example, or as inter-processor communication systems gain new capabilities – understanding and modifying program behavior will become a major bottleneck in the overall cycle of supercomputer utilization. The cost of developing these tools is likely to be less than the purchase price of any of the next generation of teraFLOPS computers, and the overall return much greater.

1 P. Bates and J. Wileden. High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *Journal of Systems and Software*, 3(4):225–264, April 1983.

2 W. Brantley, K. McAuliffe, and T. Ngo. RP3 Performance Monitoring Hardware. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 35–48. ACM Press, 1989.

3 D. Callahan and K. Kennedy. Compiling Programs for Distributed Memory Multiprocessors. *International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

4 A. Couch and D. Krumme. Projection, Pursuit, and the Triplex Tool Set for the NCUBE Multiprocessor. Technical report, Tufts University, Department of Computer Science, November 1989.

5 T. DeFanti, M. Brown, and B. McCormick. Visualization: Expanding Scientific and Engineering Research Opportunities. *IEEE Computer*, 22(8):12–26, August 1989.

6 G. Fox et al. Fortran D Language Specification. Technical Report 90-141, Rice University, Department of Computer Science, December 1990.

7 G. Geist et al. PICL: A Portable Instrumented communication library, C Reference Manual. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, 1990.

8 S. Hiranandani et al. An Overview of the Fortran D Programming System. In *Languages and Compilers for Parallel Computing, Fourth International Workshop*. Springer-Verlag, 1992.

9 C. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*, University of Queensland, February 1988.

10 K. Frenkel. The Art and Science of Visualizing Data. *Communications of the ACM*, 21(2):110–121, February 1988.

11 Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

12 M. Heath and J. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, September 1991.

13 S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

14 C. Kilpatrick, K. Schwan, and D. Ogle. Using Languages for Describing Capture, Analysis, and Display of Performance Information for Parallel and Distributed Applications. In *Proceedings of the 1990 International Conference on Computer Languages*, pages 180–189, March 1990.

15 C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting Shared Data Structures on Distributed Memory Architectures. Technical Report 90-7, Institute for Computer Applications in Science and Engineering, January 1990.

16 J. Kohl and T. Casavant. Use of PARADISE: A Meta-Tool for Visualizing Parallel Systems. In *Proceedings of the 5th International Parallel Processing Conference*, pages 561–567, May 1991.

17 L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

18 M. LaPolla. Towards a Theory of Abstractions and Visualizations for Debugging Massively Parallel Programs. In *Proceedings of the 25th Hawaiian International Conference on System Sciences*, January 1992.

19 J. Larson. Cray X-MP Hardware Performance Monitor. *Cray Channels*, pages 18–19, Winter 1986.

20 T. LeBlanc and J. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987.

21 J. Lee and D. Gannon. Object Oriented Parallel Programming: Experiments and Results. In *Supercomputing '91*, pages 273–282. IEEE Computer Society and ACM SIGARCH, November 1991.

22 A. Malony. Virtual High-Resolution Processing Timing. Technical Report CS-616, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1986.

23 A. Malony. *Performance Observability*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, September 1990.

24 A. Malony, D. Hammerslag, and D. Jablonowski. TraceView: A Trace Visualization Tool. *IEEE Software*, 8(5):19–28, September 1991.

25 A. Malony and D. Reed. A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 213–226, Amsterdam, The Netherlands, 1990. Association for Computing Machinery.

26 A. Malony and D. Reed. Models for Performance Perturbation Analysis. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 1–12. ACM Sigplan/Sigops and Office of Naval Research, May 1991.

27 A. Malony, D. Reed, J. Arendt, R. Aydt, D. Grabas, and B. Totty. An Integrated Performance Data Collection Analysis, and Visualization System. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 229–236. Association for Computing Machinery, March 1989.

28 A. Malony and E. Tick. Parallel Performance Visualization, February 1991. A proposal to the National Science Foundation.

29 G. Meyer and D. Greenberg. Perceptual Color Spaces for Computer Graphics. In H. John Durrett, editor, *Color and the Computer*. Academic Press, 1987.

30 B. Miller, M. Clark, S. Kierstead, and S. Lim. IPS-2: The Second Generation of a Parallel Program Measurement System. Technical Report CS-783, University of Wisconsin at Madison, Department of Computer Science, August 1988.

31 A. Mink and R. Carpenter. A VLSI Chip Set for a Multiprocessor Performance Measurement System. In M. Simmons, R. Koskela, and I. Bucher, editors, *Parallel Computer Systems: Performance Instrumentation and Visualization.* ACM Press, 1990.

32 A. Mink and G. Nacht. Performance Measurement of a Shared-Memory Multiprocessor using Hardware Instrumentation. In *Proceedings of the Twenty-Second Hawaii International Conference on System Sciences*, 1989.

33 D. Reed, R. Olson, R. Aydt, T. Madhyastha, T. Birkett, D. Jensen, B. Nazief, and B. Totty. Scalable Performance Environments for Parallel Systems. Technical Report UIUCDCS-R-91-1673, University of Illinois at Urbana-Champaign, Department of Computer Science, March 1991.

34 D. Reed and D. Rudolph. The Intel iPSC/2: An Approach to Performance Instrumentation. *International Journal of High Speed Computing*, pages 517–542, December 1990.

35 M. Reilly. *A Hybrid Performance Monitor for Parallel Programs.* Academic Press, 1990. Ph.D. dissertation, Carnegie-Mellon University, Department of Electrical and Computer Engineering.

36 P. Robertson and J. O'Callaghan. The Application of Scene Synthesis Techniques to the Display of Multidimensional Image Date. *ACM Transactions on Graphics*, 4:247–275, 1985.

37 D. Rover and C. Wright. Pictures of Performance: Highlighting Program Activity in Time and Space. In *Proceedings of the 5th Distributed Memory Computing Conference*, April 1990.

38 S. Sarukkai and D. Gannon. Parallel Program Visualization using SIEVE, October 1991. Presented at the Workshop on Parallel Computer Systems: Performance Tools.

39 Z. Segall and L. Rudolph. PIE: A Programming and Instrumentation Environment for Parallel Processiong. *IEEE Software*, 2(6):22–37, November 1985.

40 M. Simmons, R. Koskela, and I. Bucher, editors. *Instrumentation for Future Parallel Computing Systems.* ACM Press, 1989.

41 M. Simmons, R. Koskela, and I. Bucher, editors. *Parallel Computer Systems: Performance Instrumentation and Visualization.* ACM Press, 1990.

42 R. Snodgrass. A Relational Approach to Monitoring Complex Systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

43 E. Tick and D.-Y. Park. Kaleidescope Visualization of Fine-Grain Parallel Programs. In *Proceedings of the 25th Hawaiian International Conference on System Sciences*, January 1992.

44 Gregory V. Wilson and Felicity A. W. George. Using Dynamic Programming to Benchmark Communications on Parallel Computers. In *Proceedings of the Fifth North American Transputer Users' Group.* IOS Press, 1992.