Data Interpretation and Experiment Planning in Performance Tools

Moderator: Allen D. Malony
University of Oregon.

Session Abstract

The parallel scientific computing community is placing increasing emphasis on portability and scalability of programs, languages, and architectures. This creates new challenges for developers of parallel performance analysis tools, who will have to deal with increasing volumes of performance data drawn from diverse platforms. One way to meet this challenge is to incorporate sophisticated facilities for data interpretation and experiment planning within the tools themselves, giving them increased flexibility and autonomy in gathering and selecting performance data. This panel discussion brings together four research groups that have made advances in this direction.

A Formal Theory of Performance Diagnosis Processes

B. Robert Helm
University of Oregon

We are currently developing performance analysis tools for a portable, scalable, high-level programming language, pC++. The features of this language have raised some fundamental challenges for our research:

- Because the language implementation is *scalable*, programs in pC++ might run on anywhere from one processor to thousands, with runtimes ranging from milliseconds to days. How can we help programmers collect enough data, but not too much, over such a wide range of program scales?
- Because programs in pC++ are *portable*, they might run on numerous platforms, from small-scale shared-memory machines to large-scale distributed systems. Each platform has its own characteristic performance concerns and measurement characteristics. How can we help programmers select useful performance metrics and displays over such a diverse range of architectures?

We do not believe any single data collection and analysis strategy can meet these challenges. What we require instead is a formal theory of *performance diagnosis processes*. Such a theory should describe the strategies by which programmers assess and explain performance problems. The theory should predict where a strategy will work well, and where it will not. Formally encoded in a performance tool, the theory could help programmers plan efficient experiments, and configure the tool to support those experiments.

We have developed a theory of parallel performance diagnosis processes, based on formal models of diagnosis developed for other fields. We have used our theory to describe performance diagnosis case studies involving both performance tool developers and scientific programmers. The results of these studies show that the theory is useful for describing performance diagnosis methods, and for evaluating performance tool features. The theory and the results of these studies are being incorporated into Poirot, a toolkit for managing performance diagnosis.

SIGMETRICS '95, Ottawa, Ontario, Canada

Online Semi-Automatic Performance Debugging

Jeffrey K. Hollingsworth
University of Maryland

Large parallel and distributed systems are being built to meet the computing needs of scientific applications and commercial database systems. Due to their complexity, it is not possible to completely model and analyze these systems off line. Instead, it is vital to monitor a running system and understand what performance problems exist. However, due to the rate of computation and communication, monitoring all aspects of every processor and communication link is not practical. Even monitoring relatively small systems today can generate gigabytes of data every second. But to understand the performance of these large systems, we need to collect sufficiently detailed information to isolate the problem. To handle this dilemma requires new approaches to performance measurement.

We have been developing a new approach to performance monitoring called Dynamic Instrumentation, which differs from traditional data collection because it defers selecting what data to collect until the program is running. This permits insertion and alteration of the instrumentation during program execution. It also features a new type of data collection that combines the low data volume of sampling with the accuracy of tracing. Instrumentation to precisely count and time events is inserted by dynamically modifying the binary program. These counters and timers are then periodically sampled to provide intermediate values to higher level consumers of performance data. Based on this intermediate data, changes are made in the instrumentation to collect more information to further isolate the bottleneck.

While tool builders work to efficiently collect data, users are being inundated with mounds of statistics that require a performance expert to interpret. The are several ways to manage this information overload problem. First, we need monitoring techniques that are demand driven rather than supply driven. Just because we can collect some bit of data doesn't (necessarily) mean we *should* collect it. Second, as the data is collected, we must help users to sort through it to identify performance problems. We have been developing a strategy called the W^3 Search System to provide users with answers to their performance questions. The goal is to develop performance measurement tools that assist programmers by automating the search for performance problems.

When looking for a performance problem by hand, programmers (generally) start from a high level view of their application and iteratively isolate the source of a performance problem to more specific components of their application. The W^3 Search System provides this same approach by iteratively answering three questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. To answer the *why* question, tests are conducted to identify the type of bottleneck (e.g., synchronization, I/O, computation). Answering the *where* question isolates a performance bottleneck to a specific resource used by the program (e.g., a disk system, a synchronization variable, or a procedure). Answering *when* a problem occurs, tries to isolate a bottleneck to a specific phase of the program's execution.

We have built an initial implementation of Dynamic Instrumentation and the W^3 Search System as part of the Paradyn Parallel Performance Monitoring Tools. Several real parallel application

programs have been measured using the system, and the results have been encouraging. First, for each of the applications studied, several significant performance bottlenecks have been identified. Second, the volume of data gathered has been two to three orders of magnitude less than event logging (tracing) would generate. Third, the perturbation of application execution time has been under 10%.

Performance Measurement for High-Level Parallel Programming Languages

Barton P. Miller
University of Wisconsin

High-level parallel programming languages promise to make programmers' lives easier. They offer portable, conceptually compact notations for specifying parallel programs, and their compilers automatically map programs onto complex parallel machines, freeing programmers from the difficult, error-prone, and sometimes ineffective task of specifying parallel computations explicitly. Unfortunately, effective performance measurement tools for high-level parallel programming languages are difficult to build because they must account for implicit low-level activities created by compilers and must present performance information about those activities in terms of the source code language.

We argue that performance measurement (and other) tools must support two basic features: (1) performance information that is relevant to the high-level source code, and (2) the ability to peel back layers of abstraction to examine low-level problems while maintaining references to the high-level source code that ultimately caused the problem. The first point is generally agreed upon, and we feel that the second point is equally as important. We make this point because while high-level language abstractions hide the gritty details to speed the building of parallel program, these abstractions also hide the causes of performance problems.

We have built a facility into our Paradyn Parallel Performance Tool that can present performance information at multiple layers of abstraction. In Paradyn, a level of abstraction includes a collection of code and data objects, and the operations on these objects. Performance data may be presented in terms of obvious high-level constructs, such as parallel loops, or less obvious ones, such as parallel arrays. Performance data can be view in terms of these parallel constructs, or in terms of the primitive computation and communication operations. Our prototype implementation can handle data-parallel Fortran (CM Fortran) and has been used to make significant performance improvements in real applications.

Interactive Parallel Programs: The On-line Monitoring and Steering of High Performance Codes

Karsten Schwan
Georgia Institute of Technology

Our group is constructing and experimenting with parallel programs that simultaneously run on a variety of parallel machines linked with high performance networks. The primary purpose of our research is to exploit these machines' capabilities to offer human-interactive interfaces that can execute simultaneously with a program's computational and communication tasks. Specifically, we are exploring the potential for increases in performance and functionality gained by the on-line interaction of end users with their supercomputer applications on single and on networked parallel machines. Therefore, we are concerned with human or algorithm interactions with the programs themselves, called "interactive program steering".

The program steering addressed by our work encompasses rapid, on-line program changes that require built-in, customized monitoring algorithms determining and enacting such changes. It also includes gradual changes to long-running real-time programs and scientific applications that interact at human speeds via user interfaces. In any case, program steering is based on the on-line capture of information defining current program and configuration state, and it assumes that human users or algorithms inspect such information and manipulate it to make steering decisions.

Our current work focuses on the opportunities and costs of on-line monitoring and steering with two substantial parallel applications: (1) a molecular dynamics simulation MD used by physicists for exploration of elementary properties of lubricants, and (2) an atmospheric modeling code used by scientists for global atmospheric modeling. We demonstrate the potential of improving performance by on-line steering with initial experimental results attained on shared memory multiprocessors. An experimental system for the on-line monitoring and steering of parallel programs -- the Falcon system -- is used in interactive runs of the MD and atmospheric codes. Using Falcon can improve program performance substantially because it allows users to manipulate the programs' data decompositions in order to balance the loads among different parallel execution threads. Using Falcon can also improve the effectiveness of end users and developers alike because it allows both to create, use, and inspect visual presentations of program output along with animated displays of program performance information. Such information is gathered via Falcon's monitoring system, which enables the on-line capture, analysis, and display of the program information that is required for program steering.

New research topics must be addressed in order to enable and facilitate on-line monitoring and steering. First, monitoring and steering must be performed in the abstractions familiar to end users, such as 'global energy', 'particle locations', etc. Such application-specific monitoring is neither easy to implement nor readily available in current performance-oriented program monitoring systems. Second, since one purpose of program steering is the on-line improvement of program performance, not only on-line monitoring must be 'cheap' relative to the frequency of steering actions, but also, more importantly, monitoring overheads must be controlled and contained throughout program execution. This implies that the monitoring system itself must be steered in terms of the amounts of information being collected, the frequencies of information collection, etc. Third, for on-line program steering, it is typically important to reduce or at least control the latency between event occurrence in the program and event processing or visualization by the monitoring system. This requires mechanisms that differ from the event files or trace files employed in current systems. Last but not least, many issues exist with respect to the on-line presentation of program information, using visual or even textual interfaces. For instance, the order in which the program information is produced by the monitoring system typically does not correspond to the order in which program events occur. Therefore, on-the-fly event reordering becomes necessary. This also implies that on-line information analysis cannot always employ existing implementations of statistical analysis tools or of graphical display tools, due to the misordering of events being received and due to possible incompleteness of information.

To realize on-line program steering, it is also important to support end users in performing steering tasks in terms of (1) models of the parallel program being constructed, typically available in part from compilers and other program construction tools and/or (2) models of the parallel program's execution and functionality with which end users are familiar, often presented by data and program visualization tools. Our research is addressing both, by construction of an interactive program tuning tool capitalizing on program information available from compilers, and by construction of interactive data visualizers and manipulators, again using the aforementioned MD and atmospheric modeling codes.