Call for Collaboration: Performance Diagnosis Processes

Allen D. Malony and B. Robert Helm
University of Oregon

CIS-TR-95-01
February 1995

Abstract

Our research focuses on the location and explanation of performance problems in parallel programs, a task that we call *performance diagnosis*. Researchers have developed many software tools to collect and analyze data for performance diagnosis, but many obstacles prevent such tools from practically benefitting parallel programmers. Two obstacles in particular motivate our current work:

1. Researchers lack a theory of what methods work, and why. There is no formal way to describe or compare the ways expert programmers solve their performance diagnosis problems in particular contexts.

2. Parallel programmers lack a guide to what tools work, and where. There is no standard framework for understanding tool features and fitting them to the programmer's particular needs.

Both of these obstacles, we believe, could be mitigated by a formal theory of performance diagnosis processes. This article summarizes such a theory, and proposes a research collaboration to evaluate that theory against actual programming practices.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Call for Collaboration: Performance Diagnosis Processes

Our research focuses on part of the parallel programmer's task that we call performance diagnosis. Parallel programmers often improve program run time enormously by running experiments on a target machine, and using their results to guide algorithm design changes and code fixes. The programmer makes experimental runs on a machine either to find performance problems in a single version of the program (performance debugging), or to find out and explain the relative performance of multiple versions (performance comparison). In either type of experiment, one frequently seeks the principal sources of overhead or "bottlenecks": program behavior that limits the program's performance on the machine. Performance diagnosis is the task of finding such bottlenecks.

Researchers have developed many software tools to collect and analyze data for performance diagnosis. However, parallel programmers generally do not use these tools. Many obstacles prevent performance diagnosis tools from escaping research into practice. Two in particular motivate our current work:

1. Researchers lack a theory of what methods work, and why. There is no formal way to describe or compare the ways expert programmers solve their performance diagnosis problems in particular contexts. Consequently, there is no method to predict what will work in a new context.
2. Parallel programmers lack a guide to what tools work, and where. There is no standard framework for understanding tool features and fitting them to the programmer's particular needs. As a result, potential tool users cannot locate tools to solve their performance diagnosis problems.

Both of these obstacles, we believe, could be mitigated by a formal theory of performance diagnosis processes. Such a theory should be able to describe the problem-solving carried out by programmers, and the problem-solving supported by tools. The theory should be able to predict situations where a performance diagnosis method will work well, and situations where it will not. Researchers could use the theory to create more effective performance tools, and parallel programmers could use the theory to select effective performance tools. Ideally, the theory could be extended beyond parallel program performance to other areas of software performance engineering.

We have made a very modest first step toward such a theory. We have evaluated our theory by using it to describe case studies in papers by performance tool developers. However, as many have observed, there are many differences between tool development and programming for computational science. To test our theory more realistically, we need to study the performance diagnosis practices of people who actually program scientific computations. We therefore are seeking case studies, testimonials, horror stories, and any other formal and informal data from computational scientists and programmers on how they locate and explain performance problems. We are also seeking informants willing to take part in a structured interview on their methods of performance diagnosis.

The study will be conducted initially by e-mail over a period of one to two months. The goal will be a published report on the theory and the results of the study. Participants will receive co-author credit in the report. If initial results seem promising we may wish to arrange visits to one or more sites for more formal field observation.

Interested persons should contact:

Dr. Allen D. Malony (malony@cs.uoregon.edu)
B. Robert Helm (bhelm@cs.uoregon.edu)
Department of Computer and Information Science
University of Oregon
Eugene, OR 97043
Tel: (503) 346-4426
Fax: (503) 346-5373

1.0 Motivation

The substance of our research, is to develop and evaluate a theory of expert parallel performance diagnosis in scientific computing, as described in the attached "Call for collaboration". We have developed a preliminary theory of performance diagnosis that will appear in a forthcoming report. That theory, however, has two limitations:

1. The theory is based on case studies in performance tool papers, and therefore may not be adequate to capture the performance diagnosis behavior of scientific parallel programmers, the group whose needs motivate the research.
2. The theory is weak on decision-making. That is, it does not identify criteria for selecting a particular performance diagnosis strategy.

Both shortcomings, we believe have to be at least partially fixed for our research. We need to ground the theory in some "realistic" case studies to further develop and eventually evaluate the theory. We need a more developed decision framework to make the theory apply practically to actual performance diagnosis cases. We have therefore have sent out the attached "Call for collaboration", seeking participants for additional case studies in performance diagnosis.

In what follows, we very briefly explain the theory as it now stands. We then identify why and how we would like to work with the application developers. The material in section 2 (the theory) will appear in a more complete form in the aforementioned report.

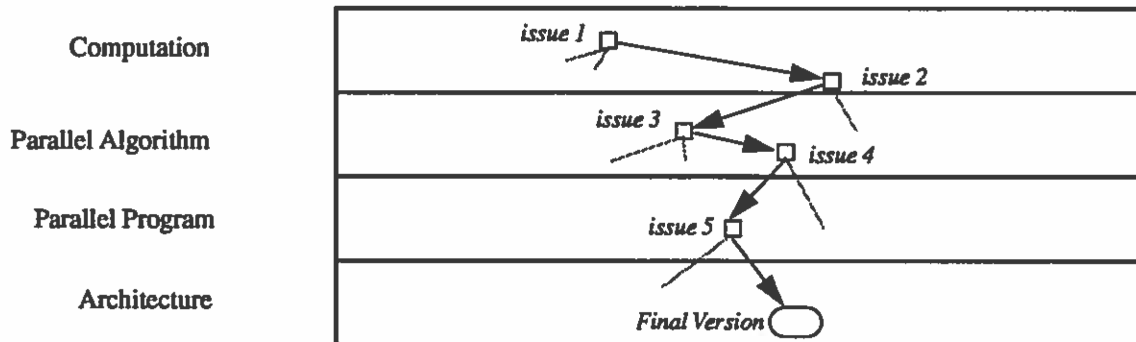
2.0 Overview of a theory of performance diagnosis

The theory is based on a simple model of scientific parallel programming as mapping across system description levels (Figure 1). The programmer begins with a mathematical description of the scientific computation to perform. For many problems, this description will be a set of partial differential equations and boundary conditions. This description is mapped to a parallel algorithm, which is coded in a parallel language and compiled for the parallel architecture. Alternatively, it is mapped to a serial algorithm and program from which the compiler develops a parallel algorithm, parallel program, and executable.

At each level of mapping, the programmer must resolve a set of mapping issues. An issue is a decision a programmer must make to map between two levels. For instance, to map to the algorithm level, the programmer must decide which kinds of parallelism from the computation to use on the parallel machine. To map to the parallel machine level, the programmer might have to decide how to assign physical processors to virtual processes or threads in the program. Each parallel program version therefore has an associated issue structure, consisting of the issues addressed and the way that they were resolved. We do not claim that the programmer is aware of this structure in its entirety; one will often resolve an issue without even

explicitly formulating it. However, performance problems will bring issues forgotten or overlooked to the surface.

FIGURE 1. A simple model of parallel programming,



Performance diagnosis, we conjecture, is the use of experimental data to find bad mapping choices, by finding program features that lead to poor performance. The programmer diagnoses the problematic program features, then reconstructs or rationalizes the mapping choices and issues that led to those features. The diagnosis step itself comprises three activities:

1. *Experimentation.*

   The programmer experiments with of one or more executable program variants and collects performance data.

2. *Assessment.*

   The programmer assesses the collected performance data and identifies one or more key performance bottlenecks.

3. *Explanation.*

   The programmer explains the key performance bottlenecks, in terms of program features at various levels of system description.

We do not assume that these activities occur in strict sequence. Rather, they overlap and interact with one another. For instance, the programmer might do an experiment driven by a preliminary assessment of the program's bottlenecks, in order to tie down those bottlenecks more precisely. We refer to the way that a programmer sequences and overlaps performance experimentation, assessment, and explanation as that programmer's performance diagnosis *strategy.*

A theory of performance diagnosis must therefore tell how programmers do experimentation, assessment, and explanation of performance data, and how they organize those activities into a coherent strategy. We have begun to develop such a theory, by analyzing case studies using theories of diagnosis developed for other fields (such as medicine). In brief, we conjecture that assessment (process 2) is carried out by a process of "heuristic classification". The programmer tries to verify that the program has one or more predefined classes of performance bottle-

necks. The bottlenecks looked for depend mainly on the target parallel machine and language, but may also depend on program or algorithm features. The classification process is uncertain ("heuristic"), because it is difficult to tell how significant a given bottleneck is from limited performance data. Experimentation (process 1), supports classification by finding evidence for and against particular bottleneck classes in program runs.

To take an example: assume a programmer is implementing a computation in compiler-parallelized Fortran on a shared-memory multiprocessor. We conjecture that the programmer has a predefined set of bottleneck classes to look for on this platform, such as "load imbalance", or "long serial sections". The programmer experiments with the program in ways that will rule in certain bottleneck classes and rule out others. For example, a programmer attempts to fit the program's behavior to an Amdahl's law model in order to rule in "long serial sections".

While bottlenecks may be assessed by simple classification, the theory assumes that more complex reasoning is needed for process 3, explanation. We conjecture that programmers use mental simulation of a suspect program section to verify that it can account for observed bottlenecks. For example, in response to an observed load imbalance in a particular parallel section, the programmer uses a model of that section's scheduling discipline to verify that it can produce the imbalance. This simulation process can be helped by additional experimentation, in particular by visualizations of actual program behavior.

The theory is weak, at this point, on strategy: how does the programmer decide what to do in diagnosis at any point? The programmer's overall diagnosis strategy we assume, is determined by continually weighing the cost of additional analysis, versus the benefits of improved performance and confidence. The principal cost is experimentation, which consumes the most programmer and machine time. We therefore assume that the programmer frequently evaluates the cost of possible experiments. These costs are proportional to the number of runs, the expected run time and analysis, and a host of other factors yet to be determined. The costs are weighed against the potential benefits of performance improvement and insight, which can be estimated by idealized models (such as linear speedup), by severity of performance bottlenecks, or by other means yet unknown.

3.0 Proposed collaboration with application developers.

The theory sketched above is based on case studies drawn from performance tool papers. These studies have several advantages: they are easy to get, they describe the performance data collected, and they explain the reason for each step taken. On the other hand, the community of performance tool producers is not always well-connected to the community of (potential) tool consumers. As a result, we can not be certain that we have analyzed representative cases. In particular, as we indicate above, these studies have failed to identify a good set of decision criteria for choosing a diagnosis strategy. Each case that we have studied has a tool to

sell, and justifies that tool's diagnosis strategy on particular criteria. It does not follow that computational scientists and programmers use similar criteria.

Computational scientists are more likely to present representative case studies in their research papers. However, such papers sometimes omit crucial information. They sometimes present performance data to evaluate a particular code, while omitting discussion of inferior versions of that code developed along the way. Unfortunately, our theory of performance diagnosis focuses precisely on those inferior versions, and in particular on the analysis process that eliminated them. So we hope to work directly with the producers of each scientific parallel program, to ensure that we acquire not only the program's justification, but its history. We therefore are proposing to collect "program histories" for a set of case studies, to evaluate and extend our current theory of diagnosis. We believe that useful histories can be collected by e-mail and other remote means in a series of stages. The stages closely follow the model of parallel programming and the theory outlined in section 2.

1. Describe current results.

   In this stage, we will obtain a summary of the results from each project. In most cases, this will simply be the research publications, plus updates as necessary. Questions examined in this phase include "what computational problem is being solved?", "What major issues arose in mapping it to a parallel computer", and "What were the results?"

2. Identify diagnosis problems.

   In this stage, we will work with the investigators to identify the history of the programming project. In effect, we will try to find major "branch points" in the development of the current code or codes, and obtain a summary of why a particular branch was taken. Each branch point, we believe, will induce a diagnosis problem that the investigators solved. Questions examined include "How did the current version of the program differ from the initial version?", "What variants of the program/algorithm were investigated?", and "What major changes were made to the program and when?"

3. Detail diagnosis process.

   In this stage, we will work with investigators to describe the diagnosis process that was followed at each major branch point. We will try to obtain a reasonably detailed description of the performance studies that were done, and the conclusions that were reached, that led to new choices on major issues. We will use this information to check the "fit" with our current diagnosis theory, possibly finding additional gaps or failures in that theory. Questions examined include "What experiments were done to evaluate this variant of the program?", and "What bottleneck motivated that change?"

4. Analyze decision criteria.

   In this stage, we will try to identify why investigators chose to follow a particular diagnosis process. In particular, we will try to elicit the criteria, if any, that were used to rule out particular performance experiments or to decide that no further experiments were necessary.

We anticipate that this stage will be by far the most difficult. Questions include "What program measurement facilities do you habitually use among those available at your site?", and "How long were you prepared to wait for the results of that performance experiment?"

The end product of this work will be a joint report on the results: a summary of the histories obtained, and the implications of those histories for a theory of performance diagnosis. We believe such a report would be a valuable addition to the field of parallel programming. The report would provide a detailed, formal description of how performance diagnosis was done on important computational science projects. This in turn could lead to the development of more productive, usable performance tools. The theory could also serve as a formal evaluation "checklist", allowing scientific parallel programmers to compare existing and future performance tools.