MODEL-BASED AUTOMATIC PERFORMANCE DIAGNOSIS OF PARALLEL COMPUTATIONS

by

LI LI

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

March 2007

MODEL-BASED AUTOMATIC PERFORMANCE DIAGNOSIS OF PARALLEL COMPUTATIONS

by

LI LI

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

March 2007

"Model-based Automatic Performance Diagnosis of Parallel Computations," a dissertation prepared by Li Li in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

Allen D. Malony

Dr. Allen D. Malony, Chair of the Examining Committee

2-27-07

Date

Committee in charge:

Dr. Allen D. Malony, Chair
Dr. Stephen Fickas
Dr. Virginia Lo
Dr. Daniel Steck
Dr. Xian-He Sun

Accepted by:

Dean of the Graduate School

An Abstract of the Dissertation of

Li Li for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken March 2007

Title: MODEL-BASED AUTOMATIC PERFORMANCE DIAGNOSIS OF PARALLEL COMPUTATIONS

Approved: Allen D. Malony

Dr. Allen D. Malony

Scientific parallel programs often undergo significant performance tuning before meeting their performance expectation. Performance tuning naturally involves a diagnosis process – locating performance bugs that make a program inefficient and explaining them in terms of high-level program design. Important performance measurement and analysis tools have been developed to support the performance analysis with the facilities of running experiments on parallel computers and generating measurement data to evaluate performance. However, current performance analysis technology does not yet allow for associating found performance problems with causes at a high-level program abstraction. Nor does it support the performance diagnosis process in a well automated manner.

We present a systematic method to guide the performance diagnosis process and support the process with minimum user intervention. The motivating observation is that performance diagnosis can be greatly improved with the use of performance knowledge

about parallel computation models. We therefore propose an approach to generating performance knowledge for automatically diagnosing parallel programs. Our approach exploits program execution abstraction and parallelism found in computational models to search and explain performance bugs. We identify categories of knowledge required for performance diagnosis and describe how to derive the knowledge from computational models. We represent the extracted knowledge in a manner such that performance inferencing can be carried out in an automatic manner.

We have developed the *Hercule* automatic performance diagnosis system that implements the model-based diagnosis strategy. In this dissertation, we present how Hercule integrates the performance knowledge into a performance analysis tool and demonstrate the effectiveness of our performance knowledge engineering approach through Hercule experiments on a variety of parallel computational models. We also investigate compositional programs that combine two or more models. We extend performance knowledge engineering to capture the interplay of multiple models in an integrated state, and improve Hercule capabilities to support the compositional performance diagnosis. We have applied Hercule to two representative scientific applications, both of which are implemented with compositional models. The experiment results show that, requiring minimum user intervention, model-based performance analysis is vital and effective in discovering and interpreting performance bugs at a high level of program abstraction.

CURRICULUM VITAE

NAME OF AUTHOR: Li Li

PLACE OF BIRTH: Anhui Province, P.R. China

DATE OF BIRTH: June 24, 1976

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Institute of Software, Chinese Academy of Sciences, Beijing, China
Nanjing University, Nanjing, China

DEGREES AWARDED:

Doctor of Philosophy, 2007, University of Oregon
Master of Science, 2004, University of Oregon
Master of Science, 2000, Institute of Software, Chinese Academy of Sciences
Bachelor of Science, 1997, Nanjing University

AREAS OF SPECIAL INTEREST:

Performance tuning of parallel programs

PROFESSIONAL EXPERIENCE:

Research Assistant, Department of Computer and Information Science, University of Oregon, 2001-2006
Research Assistant, Institute of Software, Chinese Academy of Sciences, 1997-2000

PUBLICATIONS:

L. Li and A. D. Malony. "Automatic Performance Diagnosis of Parallel Computations with Compositional Models". Accepted by *the 12th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2007

L. Li and A. D. Malony. "Knowledge Engineering for Automatic Parallel Performance Diagnosis", *To appear in Concurrency and Computation: Practice and Experience*, 2007

L. Li and A. D. Malony. "Model-based Performance Diagnosis of Master-worker Parallel Computations." in *proceedings of Euro-Par 2006*, vol. 4128 of *Lecture Notes in Computer Science*, pages 35-46, Springer Berlin / Heidelberg, 2006

L. Li, A. D. Malony, and K. Huck. "Model-Based Relative Performance Diagnosis of Wavefront Parallel Computations", in *proceedings of High Performance Computing and Communications 2006*, vol. 4192 of *Lecture Notes in Computer Science*, pages 200-209, Springer Berlin / Heidelberg, 2006.

A. D. Malony, S. Shende, R. Bell, K. Li, L. Li, and N. Trebon. "Advances in the TAU Performance System," in *Performance Analysis and Grid Computing* (V. Getov, M. Gerndt, A. Hoisie, A. Malony, and B. Miller, eds.), pp. 129-144, Kluwer, Norwell, MA, 2003.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my advisor Allen Malony, who consistently inspired and supported me in the past five years. I will not forget his encouragement and confidence in me in the periods when confusion and frustration were great. I greatly appreciate his guidance and advice, which have made an enormous contribution to this dissertation.

I thank the members of my committee, Stephen Fickas, Virginia Lo, Daniel Steck, and Xian-He Sun, for invaluable suggestions and feedback.

Special thanks go to Dr. Zena Ariola and Virginia Lo. They set extraordinary examples of successful women in computer science. They keenly offered suggestions and assistance along the way, and their help is greatly valued and appeiciated.

I would not have completed this dissertation without my family's love, support, and patience over the long journey. I dedicate this dissertation to my grandparents, who made this dissertation possible at its very beginning. I was not able to go back to China to take care of my grandma when she was badly ill. I hope this dissertation will make her smile, just like she lightened up my life when I was a little girl.

I also dedicate the dissertation to my mentor, my best friend, my husband Kai. I would thank him for growing up with me, and going through all this with me over the years. His unwavering support and tolerance for me to explore my life in a way different from that of traditional Chinese women helped make me the person I am, and he certainly deserves most of the credit for any accomplishments of mine. I'm grateful for our every day together.

To my grandparents and Kai.

TABLE OF CONTENTS

Chapter Page

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

Scientific parallel programs often undergo significant performance tuning before meeting performance expectations. Performance tuning naturally involves a diagnosis process – locating performance bugs that make a program inefficient and explaining them in terms of high-level program design. The process of performance diagnosis, including the generation and running of experiments, the characterization of performance properties, and the locating and interpretation of performance problems (bugs), is particularly challenging to automate because it fundamentally is an intelligent process wherein we capture and apply *knowledge* about performance problems, how to detect them (i.e., their *symptoms*), and why they exist (i.e., their *causes*).

This dissertation presents a new approach to performance analysis: model-based automatic performance diagnosis. Our approach exploits program semantics and parallelism embedded in *models* of parallel computations to search and explain bugs. We present a set of principles to engineer performance knowledge from the models and use the knowledge as the basis for building a framework to support automated performance diagnosis. The framework's function is therefore guided by expert strategies for automatic problem discovery and hypothesis testing, strategies that are captured and encoded in the performance knowledge base.

Hercule is such an automatic performance diagnosis framework that we have developed using the model-based approach. The Hercule system operates as an expert system within a parallel performance measurement and analysis toolkit. Model knowledge base is the core of the system, which fuels the diagnosis process. Hercule automates all aspects of the diganosis process, including experiment construction, performance analysis, and perfor-

mance causal inferencing. Hercule diagnosis results, which consist of a list of performance problems existing in the program and corresponding explanations, provide feedback at a high level of program abstraction so the burdens imposed on the user to map primitive performance data to computational design is greatly reduced.

Our experiments corroborate the effectiveness of designing tools using a model-based approach. We have tested Hercule on a variety of parallel programs and scientific computations, coded with singleton or compositional models. For each of the applications, Hercule provides the user with an insight about how each essential design component performs and how efficient the problem concurrency is realized among parallel ingredients. In the case that more than one model is used, Hercule additionally unravels performance effects resulting from model interactions. The Hercule feedbacks point directly to performance degrading factors in the design model, filling in the semantic gap between raw measurements with program design.

1.1 Thesis Statement

The central hypothesis of this dissertation is that performance diagnosis would benefit from knowing the computation model of a parallel program and the model knowledge can support automatic performance problem discovery and interpretation if incorporated into a performance tool.

In order to substantiate this claim, we design an approach to extracting performance information from computational models that can guide the diagnosis process with minimum user intervention. We also develop Hercule that implements automatic diagnosis with model knowledge. Providing performance interpretation in model language, Hercule eliminates the need to correlate raw performance information at the level of source code with program design at a high level of abstraction . It therefore can be used easily by inexperienced parallel programmers to improve performance.

1.2 Contributions

This thesis makes the following specific contributions:

Contribution 1. A new approach for designing performance analysis tools, which enables performance problem discovery and causal inferencing at a high level of program abstraction.

Important performance measurement and analysis tools, such as Paradyn [1], AIMS [2], and SvPablo [3], have been developed to help programmers diagnose performance problems. However, the performance feedback provided by the tools tend to be descriptive information about parallel program execution at low levels of abstraction. Even if the tools detect a specific source code location or machine resource that demonstrates poor performance, the information may lack the context required to relate the performance information to a higher-level cause. For example, a performance tool feedback may look like:

> 40% synchronization time spent between lines 15-20 because proc. 2 executes the loop 300 while proc. 3 does not.

The example interprets a program inefficiency by differentiating execution paths of involving processing units. However, it falls on the users to explain the observation and reason about causes with respect to computational abstractions used in the program and known only to them. A desirable explanation in terms of the program design and behavioral characteristics would look like:

> 40% synchronization time spent between lines 15-20 because the master, proc. 2, is running task-distributing algorithm when the worker, proc. 3, is waiting in idle.

Such performance explanations can direct to the bad program design decisions (e.g., expensive task set-up cost at the master side) and help optimize performance. Unfortunately, novice parallel programmers often lack the performance analysis expertise required for high-level problem diagnosis using only raw performance data.

We believe that the deficiencies above could be addressed by incorporating program semantics into performance analysis. We are particularly inspired by expert parallel programmers who often approach performance tuning in a systematic, empirical manner by

running experiments on a parallel computer, generating and analyzing performance data, and then testing performance hypotheses to decide on problems and prioritize opportunities for improvement. Implicit in this process is the expert's knowledge of the program's code structure, its parallelization approach, and the relationship of application parameters to performance. We therefore specify a set of program-specific information required for high-level performance interpretation (named *performance knowledge*), which includes behavioral descriptions, performance metrics, and high-level design factors. And we advocate looking to models of parallel computations as sources of the information required for diagnosis as the models abstract parallel execution details and provide a semantic basis for parallel program development. Provided with the program-specific information, it is possible for a performance analysis tool to produce feedback directly relating to program design.

Contribution 2. A model-based knowledge engineering approach that systematically acquires and represents performance information required for diagnostic analysis.

Parallel computational models attract our attention in our search for the answer to "Where does the performance knowledge come from?" Models are recurring algorithmic and communication patterns in parallel computing, and are widely used in the design of parallel programs. The models provide semantically rich descriptions that enable better interpretation and understanding of performance behavior. Our view is that we can generate basic performance knowledge from the design models, from which program-specific information will be derived.

In order to generate performance knowledge from models and use it to diagnose realistic parallel programs, we specifically identify methods for parallel model representation, performance modeling, metric definition, and performance bug search and interpretation methodology. The performance knowledge derived in this manner supports bottom-up inference of performance that starts with primitive performance data and ends up with high-level explanations. And it provides a sound basis for automating diagnosis processes. We encode and store the knowledge in a base foundation and interface it to a performance analysis system. The system can then use the knowledge base for forming problem hypothesis, evaluating performance metrics to test the hypothesis, and deciding which candidate hypothesis is most useful to pursue and new experiment requirements are generated to con-

firm or deny it. The knowledge-driven inference relaxes the requirement of intelligent input from the user.

Contribution 3. A framework that implements automatic performance diagnosis with a model-based approach.

The design of performance experiments, examining performance data from experiment runs, and evaluating performance against the expected values to identify performance bugs are not well automated and not necessarily guided by a diagnosis strategy in existing performance tools. Typically, the user decides on the instrumentation points and measurement data to collect before an experimental run. The user is also often involved with the processing and interpretation of performance results. The manual efforts required by tools and the lack of support for managing performance problem investigation ultimately limit diagnosis capability.

We have developed Hercule, an automatic performance analysis framework, that uses our model-based approach. A knowledge base that consists of model-derived performance knowledge is the core of the system. Interfacing the knowledge base to an inference engine and performance measurement toolkits would automate the laborious experiment-and-analysis process that is otherwise imposed on the user.

Given a program to be diagnosed, Hercule is informed of its computational model and then comes up with a set of experiment instructions by referring to the model knowledge base. Hercule has the measurement toolkits execute the corresponding experiments to collect necessary performance data. Hercule automates data analysis, performance inferencing, and possibly more iterations of experiments, and finally reaches conclusions about performance with respect to the model use.

To date the knowledge base in the Hercule system have included Master-Worker, Wavefront, Divide-and-Conquer, AMR, and Geometric Decomposition model knowledge. We tested Hercule on a range of parallel programs. The experiment results show that Hercule is a viable system that is able to design performance experiments and conduct bug search and causal reasoning in an automated manner. They also corroborate the usefulness of designing tools based on our model-based approach.

Contribution 4. A set of techniques to address the performance impact of model composition in the diagnosis system.

Model composition, as one of the most common means of model application, captures how singleton models are composed together and interact in a parallel programs. The challenges to diagnose compositional parallel program are that performance effects of individual models may change and new effects may arise from the composite interactions. We extend the knowledge engineering and problem inferencing to capture the interplay of one model with another. Classifying model compositions into different patterns, we present a set of guidelines for identifying performance nuances introduced by each pattern, and incorporate them into steps towards the knowledge generation, such as behavioral modeling and causal inferencing, so that the performance effects of model variations and interactions will be taken into account in the diagnosis process. The adaptations to (compositional) model implementation eventually improves the quality of performance diagnosis as demonstrated on some scientific applications.

1.3 Dissertation Overview

This dissertation is organized as follows.

Chapter II describes a new approach to performance diagnosis. The approach incorporates performance knowledge into generic iterative diagnosis processes to address the automation of performance diagnosis at a high level of abstraction. Required performance knowledge is identified and classified into four categories: experiment design and management, performance models, performance evaluation metrics, and performance factors at a high level of abstraction. Parallel computational models are examined and evaluated as a dependable source of performance diagnosis knowledge.

In Chapter III, a model-based performance knowledge engineering approach is presented. Our approach addresses how to extract expert performance knowledge from a parallel model with four types of modeling: behavioral modeling, performance modeling, model-specific metric definition, and inference modeling. We demonstrate the approach with a parallel Divide-and-Conquer model.

Chapter IV then describes how we implement model-based performance diagnosis in the Hercule system. We will discuss Hercule design issues, how to use Hercule from a user's perspective, and validation of Hercule diagnosis results. Next in Chapter V, we apply Hercule to three parallel applications that represent Divide-and-Conquer, Master-Worker, and Wavefront model, respectively. We provide Hercule results demonstrating the effectiveness of our model-based approach to performance diagnosis. We particularly extend Hercule to support relative performance diagnosis from a multi-experiment view. Understanding of performance problems routinely involves *comparative* and *relative* interpretation. Relative diagnosis contrasts performance of multiple experimental runs with varied problem sizes or process scales. It finds performance anomalies as problem scales and explains them in terms of program behaviors and design features. Relative diagnoses of Sweep3D (implemented with Wavefront model) performance anomalies in strong and weak scaling cases are presented.

In Chapter VI, we extend the model-based diagnosis methodology to support compositional models that integrate singleton computational patterns. We identify different model composition styles and discuss systematic steps to address the performance implications of model integration in performance knowledge engineering so that performance losses due to model interaction can be detected and interpreted. We enhance Hercule framework to support compositional performance diagnosis and test hercule with two scientific applications, FLASH and PDLAHQR. The experiment results are reported.

Finally, our conclusions and plans for future work are discussed in Chapter VII.

CHAPTER II

PERFORMANCE DIAGNOSIS

Performance tuning (a.k.a. *performance debugging*) is a process that attempts to find and repair performance problems (performance bugs). For parallel programs, performance problems may be the result of poor algorithmic choices, incorrect mapping of the computation to the parallel architecture, or a myriad of other parallelism behavior and resource usage problems that make a program slow or inefficient. Expert parallel programmers often approach performance tuning in a systematic, empirical manner by running experiments on a parallel computer, generating and analyzing performance data for different parameter combinations, and then testing performance hypotheses to decide on problems and prioritize opportunities for improvement. We can view performance tuning as involving two steps: detecting and explaining performance problems (a process we call *performance diagnosis*), and performance problem repair (commonly referred to as *performance optimization*). Implicit in the diagnosis process is the expert's knowledge of the program's code structure, its parallelization approach, and the relationship of application parameters to performance. Barely capturing and formalizing the expert knowledge, existing performance analysis tools provide only descriptive feedbacks about parallel program execution at low-levels abstraction and lack supports for the automatic performance reasoning.

This chapter first describes generic performance diagnosis process. Next we present a new knowledge-based approach to performance diagnosis that enables automatic performance problem discovery and causal inferencing at a high level of program abstraction and discuss the feasibility of extracting performance diagnosis knowledge from parallel computational model. We finally present literature review, contrasting our approach with other relevant research work.

2.1 Generic Performance Diagnosis Process

Performance diagnosis is the process of locating and explaining sources of performance loss in a parallel program. Expert parallel programmers often improve program performance by iteratively running their programs on a parallel computer, then interpret the experiment results and performance measurement data to suggest changes to the program. The generic diagnosis process is shown in Figure 2.1. More specifically, the process involves:

- *Designing and running performance experiments.* Researchers in parallel computing have developed integrated measurement systems to facilitate performance analysis [1, 2, 4]. They observe performance of a parallel program under a specific circumstance with specified input data, problem size, number of processors, and other parameters. The experiments also decide on points of instrumentation and what performance information to capture. Performance data are then collected from experiment runs.

- *Finding symptoms.* We define a *symptom* as an observation that deviates from performance expectation. General *metrics* for evaluating performance includes execution time, parallel overhead, speedup, efficiency, and cost [5]. By comparing the metrics computed from performance data with what is expected, we can find symptoms such as low scalability, poor efficiency, and so on.

- *Inferring causes from symptoms. Causes* are explanations of observed symptoms. Expert programmers interpret performance symptoms at different levels of abstraction. They may explain symptoms by looking at more specific performance properties [6], such as load balance, memory utilization, and communication cost, or tracking down specific source code fragments that are responsible for major performance loss [7]. Attributing a symptom to culprit causes requires bridging a semantic gap between raw performance data and higher-level parallel program abstraction. Expert parallel programmers, relying on their performance analysis expertise and knowledge about program design, are able to form mediating hypotheses, capture

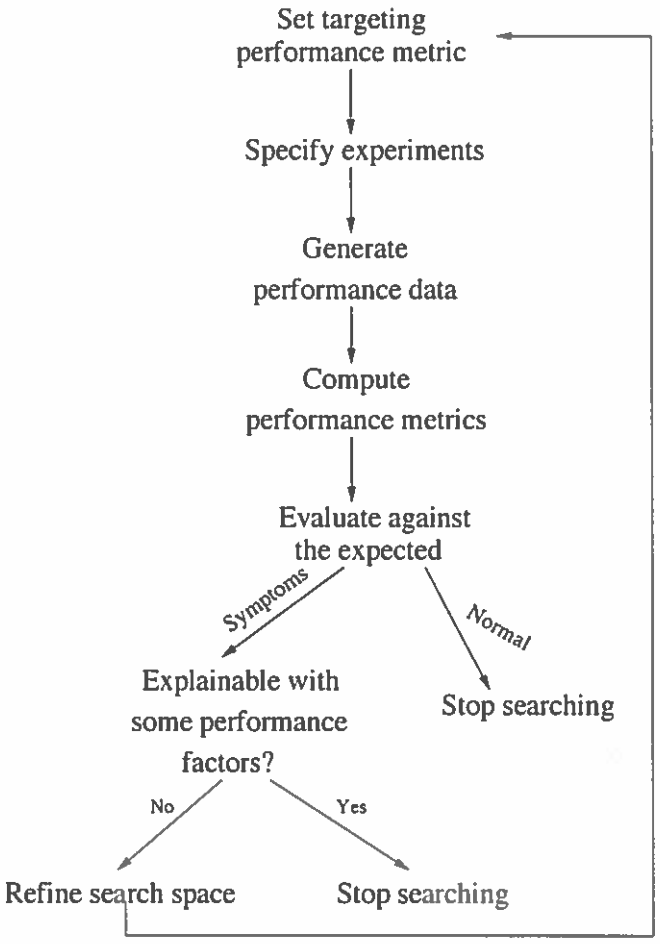


FIGURE 2.1: A high-level overview of generic iterative diagnosis process.

supporting performance information, synthesize raw performance data to testify the hypotheses, and iteratively refine hypotheses towards higher-level abstractions until some cause is found.

2.2 Knowledge-based Automatic Performance Diagnosis Approach

We believe that both of the deficiencies in existing performance analysis tools, low-level feedbacks and lack of automation support, could be addressed by encoding how expert parallel programmers debug performance problems. In particular, we want to capture

performance knowledge about program's code structure, its parallelization approach, performance problems and expert strategies for detecting them, and then apply it in a diagnosis system to guide performance inferencing.

2.2.1 Performance Knowledge

We identify four main categories of knowledge required by performance diagnosis.

Experiment design and specification. Empirical-based performance analysis relies on experiments to capture performance information. Experiment specification includes system parameter setting, instrumentation instruction, and decisions about what performance events to record. The experiment design should be particularly tailored to code structure to restrict performance data to a tractable level.

Performance models. A performance model, which is derived based on computational structure and parallelization approach of a program, presents performance compositions that can help identify overhead categories that often result from cooperations and interactions between parallel components. Refining the focus of performance modeling from overall system behavior to a specific or problematic behavioral aspect will help narrow the performance problem search while preserving a semantic context for reasoning the found problems.

Both experiment design and performance modeling are based on behavioral models of the parallel program. There therefore arises a need for behavioral descriptions of parallel programs in a format that is suitable for performance diagnosis.

Performance evaluation metrics. Performance diagnosis is driven by metric-based evaluations. A metric is a formulation of a performance aspect. Expert programmers define *metrics* describing certain performance properties of concern, compute them from raw performance data, then assess and interpret them in the context of parallel systems employed. Traditional performance analysis approaches use generic metrics without relevance to program semantics, such as *synchronization overhead* and *imperfect L2 cache behavior* in [6]. A consequence of evaluation with the generic metrics is that the users still need to attribute them to specific program design decisions. To enhance explanation power of performance metrics, we intend to incorporate program semantics into their definition. The advantage

of semantics-aware metrics over generic ones is that they can not only assess performance but help reason about the assessment with algorithmic design of the program.

Performance factors at a high level of abstraction. In our diagnosis approach, we aim to find performance causes at the level of parallelization design. For this end, we should identify design factors at this abstraction level that are most critical to performance. We investigate factors with respect to algorithmic or parallelism design that are specific to a problem-solving solution. A performance cause – an interpretation of performance symptoms in terms of these factors – can therefore immediately direct the programmer to bad design decisions.

2.2.2 Modeling Inference Steps

The three major types of actions involved in diagnosis – running experiments, computing and evaluating performance metrics to find symptoms, and explaining symptoms – proceed in an iteratively refined manner. Besides the performance knowledge, we also need a diagnostic strategy to guide performance problem search and inference steps, which, in our approach, determines how to invoke performance knowledge systematically.

We advocate a bottom-up inference approach that starts with performance problem discovery at a low level of abstraction, and then gradually boosts the abstraction level of causal reasoning by refining performance models. Specifically, the inference process begins with evaluation of a generic performance metric like efficiency or speedup. Corresponding performance experiments are conducted and the collected data is abstracted according to the metric computing rules. We then reach a symptom by evaluating the metric against the expected value or the tolerance for its severity. If the symptom can be directly interpreted by some performance factors at a high level of abstraction, then the search for performance causes resulting in the symptom is over. That is, there is an explanation for the performance problem. Otherwise, we refine performance models to restrict attention to more specific program behaviors, then define corresponding metrics to assess the overhead categories as revealed in the refined models. New experiment specifications and performance metric choices are generated as a result of refinement. They are fed into the next iteration of inference. As the refined performance models are specific to program behaviors,

the inferencing will eventually achieve an interpretation of found problems with program semantics.

2.2.3 Knowledge-based Automatic Performance Diagnosis

The application of the knowledge in performance diagnosis process is displayed in the Figure 2.2. We envision that the performance knowledge is stored in a base foundation that includes behavioral descriptions (for experiment design and performance modeling), performance models, evaluation metrics, and high-level performance factors. Each diagnosis stage retrieves necessary information from the knowledge base and uses it as guidance to generate stage results, therefore reducing the requirement of intelligent input from the user.

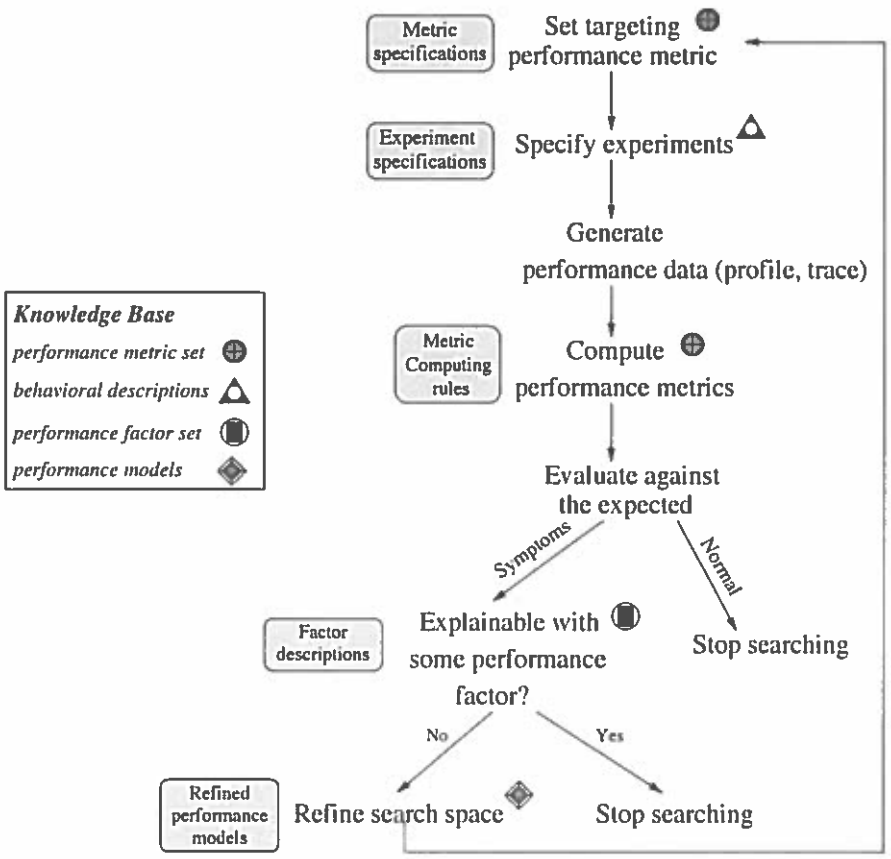


FIGURE 2.2: A high-level overview of knowledge-based automatic diagnosis process. The use of performance knowledge is annotated in the process steps.

2.3 Model-based Performance Diagnosis Approach

The answer to whether a diagnosis tool would benefit from the performance knowledge at high levels of program abstraction is most certainly "yes." The question we need to answer next is "Where does the performance knowledge come from?" Parallel computational models present possibilities to answer the question. A parallel model, also called design pattern [8, 9] or parallel programming paradigm [10] in the literature, is a recurring algorithmic and communication pattern in parallel computing, and are widely used in the design of parallel program. Typical models include master-worker, pipeline, divide-and-conquer, and geometric decomposition [9]. Representing parallelism inherent to a wide range of problems, parallel models are adopted by many realistic parallel application designs. For instance, Sweep3D [11] uses Wavefront model – a two-dimensional variant of pipeline model. SPhot [12] employs Master-Worker style of task management. Finite difference codes mostly follow geometric decomposition model.

A model usually describes computational components of a parallel program and their behaviors (algorithmic properties) and how multiple threads of execution interact and collaborate in a parallel solution (parallelism). Parallel models abstract parallelism common in realistic parallel applications and serve as a computational basis for parallel program development. It is possible to extract from them a performance knowledge foundation based on which we are able to derive performance diagnosis processes tailored to specific program implementations. Specifically, we envision that models can play an active role in the following aspects of performance diagnosis:

- ***Selective instrumentation and experiment design.*** Performance diagnosis naturally involves mapping low-level performance details to higher-level program designs, which raises the problems of what low-level information to collect and how to specify an experiment to generate the information. Parallel models identify major computational components in a program, and can therefore guide the code instrumentation and hep organize performance data produced.

- ***Detection and interpretation of performance bugs.*** In a parallel program, a significant portion of performance inefficiencies is due to process interactions arising from

data/control dependency. Parallel models capture information about computational structures and process coordination patterns generic to a broad range of parallel applications. This information provides a context for describing performance properties and attributing them to associated process behaviors. In the context of model-specific behaviors, the low-level performance details can be classified and synthesized to derive performance metrics that have explanation power at a higher level of abstraction.

- ***Expert analysis of performance problems.*** Expert parallel programmers have built up rich expertise in both programming with and analyzing commonly-used parallel models. In performance diagnosis, they implicitly refer to their prior knowledge for attributing performance symptoms to causes. Expert knowledge about the models includes model-specific performance metrics and performance factors at the level of program/algorithm design. If we can represent and manage the already available expert performance knowledge in a proper manner, they will effectively drive diagnosis process with little or no user assistance.

The above potential advantages of parallel models motivate our pursuit of a *model-based* performance diagnosis approach. Our view is that we can extract basic performance knowledge from the models, and then from which program-specific knowledge will be derived and applied to diagnosis processes. According to the diagnosis requirements, performance knowledge about a model should consist of behavioral descriptions that provide a context for performance modeling and experiment design, performance models, metrics to be used to evaluate model-specific performance aspects, and performance-critical design factors associated with the model (which we call *performance factors* and form candidate causes for interpreting performance problems). And the performance knowledge should be able to support bottom-up inference of performance causes. In next chapter, we will present a systematic approach to generating performance knowledge from models.

2.4 Related Work

Existing techniques for performance analysis and debugging focus primarily on two aspects – evaluating performance and locating problems through a defined set of performance metrics, and explaining causes of detected performance problems.

2.4.1 Property- and Metric-based Performance Bottleneck Search

Raw performance data collected through instrumentation and measurement provides little insight into understanding parallel application performance without being related back to the program and reduced into more problem-specific forms. Rather, Most existing performance debugging techniques synthesize raw data and calculate performance metrics that reflect various performance aspects at a higher level of abstraction. They then use the metrics to search for performance bottlenecks. The metrics are meaningful and closer to programmer's understanding, making it easier to interpret a found problem. Often the metrics are also associated with particular programming constructs, code regions, and processing nodes to help pinpoint performance bottlenecks in source code. In this way, performance problems that cause performance degradation in certain aspect can be tracked down to specific locations in the program.

The idea of enumerating performance properties and problems is found in a number of tools. Paradyn [1] is a performance analysis system that automatically locates bottlenecks using the W^3 search model. According to the W^3 model, searching for a performance problem is an iterative process of refining the answer to three questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. To answer the "why" question, Paradyn includes hypotheses about potential performance problems in parallel programs, and collects performance data to test whether these problems exist. The types of bottlenecks include synchronization, I/O, computation, etc. 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 isolates a problem to a specific phase of the program's execution. Each of the "why", "where", "when" axes is represented as one or more hierarchical trees, with

children nodes being the refinements or the instances of the parent nodes. The search process is conducted by the *Performance Consultant* module without requiring the user to be involved. The Performance Consultant selects a search refinement in a three-step process: determining a list of possible refinements by considering the children of the current nodes along each axis, ordering this list using internally-defined hints, selecting one or more refinements to try from the ordered list. If a selected refinement is not true, it considers the next item from the ordered refinement list. Performance bugs Paradyn targets are not in direct relation to parallel program design. It is not intended for explanation of high-level bug either.

JavaPSL [13] is an implementation of the *Performance Specification Language* (PSL) developed by the APART project [14]. PSL describes experiment-related data and performance properties of applications by using syntax and semantic rules, captured by the Java programming language in the case of JavaPSL. A performance property (e.g. load imbalance, synchronization overhead) characterizes a specific negative performance behavior of a program. Compared with common performance metrics such as execution and communication time, cache misses, performance properties provide higher level performance information, easier to interpret and compare across data collected from different programming paradigms or underlying hardware platforms. JavaPSL associates a set of primitive performance data in a user-specified code region to a specific performance property in terms of rules the user defines, and relates the property to its experiment environment. In this sense, JavaPSL works as a high-level and portable interface that enables the user to normalize and interpret performance data and to define new performance properties without knowing or changing the underlying implementation details of the tool that makes use of these properties.

There are also many other approaches to performance overhead/metric definition that follow the same design theme as JavaPSL and *performance indices*, such as *performance predicates* [15] that are computed by recognizing inefficient states and counting their duration during execution, and performance overheads that express non-scalability of a program [16].

JavaPSL is intended for flexibly defining performance bottlenecks. A bottleneck analysis tool using JavaPSL can automatically search for bottlenecks by navigating through the

performance data space and computing pre-defined performance properties. Aksum [6], for example, is such a performance analysis tool. EXPERT [17] performance tool also incorporates the notion of performance property to make performance problem definition flexible and extensible. The bottlenecks detected in terms of property definition, however, cannot be sufficiently interpreted with only the program code regions where the bottlenecks occurred (but not necessarily the cause of the problem occurred.) and information about the experiment environment. Moreover, the performance properties that are definable using JavaPSL are at relatively low semantic level in that they have little relevance to program semantics, which makes it difficult to reason about them from parallel program design point of view. In distinct to JavaPSL, we intend to define performance metrics in terms of model-semantics to enhance their explanation power.

To some extent, all the performance debugging methods above are attempting to characterize observed measurements in the form of performance properties and then match those determined properties to performance problems. The performance debugging is only as good as the quality of the properties and coverage of the problem space. The tests used to validate a problem hypothesis are expressed in the tools in terms of a threshold and one or more performance metrics. The metrics definition in Paradyn, for instance, are intended to constrain performance data to particular resource such as machines, procedures, files, communication channels (or combination of these resources), hence at low semantic level. While performance debugging tools such as Paradyn do incorporate aspects of hypothesis refinement in the search process, it is difficult for these tools to reason about problems with respect to their cause.

2.4.2 Causality Analysis

Interpreting detected performance problems requires a certain degree understanding of the parallel code. The interpretation should reveal the nature of the problems and system parameters (algorithms, systems, etc.) contributing to them. In the case that performance inefficiency occurring at a code region is caused by misbehavior in other parts of the program, it should be able to identify the relations among the participating code regions. A handful of performance analysis approaches addresses the issues about explaining perfor-

mance problems to the user. It is the aspect of explanation that is important for performance diagnosis.

Cause-effect analysis[18, 19], for example, is an automated inference process that presents explanations for dynamic phenomena of parallel program execution in terms of underlying causes. The analysis approach is centered around explaining the occurrence of a class of local events or states in terms of the events that occurred *earlier* in the execution. One example of a local state is "the processor is waiting at a synchronization point", and the corresponding explanation might describe the execution path difference between the processor and the other synchronizing processors. Two important issues involved in the approach are identifying the bounded region of events that cause a particular effect and determining the form of explanation that is enlightening to the user. With respect to the example of "a processor waiting at a barrier for another processor", cause-effect analysis might present an answer to such a question: how did the execution paths of these two processors differ since the last time they synchronized such that one processor arrived at the barrier before the other? In addition to the waiting time, cause-effect analysis is also applicable to explaining page fault events in distributed shared memory environments and transaction abortions in parallel file systems [19]. Although cause-effect analysis focuses on the inference process that leads from observed effects to root causes, its applicability is limited. First, it requires the presence of certain delimiting state/event, such as "the last time two processors synchronized", in order to make an explanation. In practice, many applications do not render such state/event explicitly. Second, the performance inefficiencies it can detect and explain in a parallel program are restricted to some forms of waiting time.

As an example of a powerful causality analysis performance tool, ATExpert [20] developed at Cray Research, Inc. uses a rule-based expert system to help users locate and interpret poor performance. ATExpert can automatically make performance observations at program, subroutine, parallel region, loop, and case level. For a given region of code, ATExpert takes as input the actual speedup, the overhead, the amount of serial time, and the number of processors. Depending on what is dominating the execution, it chooses a subset of rules. It then looks for patterns in the performance data and associates them with a list of known parallel performance problems and possible causes. AtExpert helps improve performance in two aspects. First, it identifies the serial codes that account for sig-

nificant time cost. The user can restructure the codes to exploit more parallelism. Second, given the parallelization strategy employed, it locates parallel regions that are responsible for dominant performance loss due to parallelization overhead such as start-up cost, synchronization cost, etc. In either aspect, rules are associated with the observations made for a combination of a particular code region/construct and performance metrics (speedup, relative execution time, and various performance overheads). And the rules reflect not only performance debugging expertise that guides finding performance problems and explains the nature of the problems, but also knowledge of system designers that provides insight into understanding interplay between program and underlying system. As productive as ATExpert was, its scope was restricted to underutilized loop parallelization and it required tight integration with Cray's compilers and runtime libraries.

POETRIES [21] is a performance tuning tool that takes advantage of the knowledge about the high-level structure of the application to detect and correct performance drawbacks. It builds analytical performance models based on the structures and attributes performance degradation to parameters composing the models. Our approach differs from POETRIES in that, first, it targets performance explanation and, second, it features a knowledge-based inference system that diagnoses performance in an automated manner.

Rajamony [22, 23] observed that existing performance debugging tools only *describe* the performance problem, while the onus is then placed on the user to infer the cause for the performance problems. Motivated by the observation, he developed Rx, a tool to improve the performance of explicitly parallel shared-memory programs in a sequential consistent system. Specifically, Rx automatically analyzes run-time data to derive feedback and correlates the feedback with source program to facilitate reasoning about performance at the source level. Rx targets inter-process synchronization and data communication, two significant sources of overhead in shared-memory applications. It identifies excess synchronization based on conflicting-access analysis. Rx presents a complete set of sharing-data read and write access conditions under which barriers can be removed or weakened. It identifies the dependences enforced by all pair-wise synchronizations presented in the program, and extracts the minimal set of the synchronizations that enforce all the dependences so as to remove redundant ones; Rx introduces *aggregation* and *vectorization* – two transformations that eliminate certain critical sections. It also provides two ways of computation restruc-

turing which enhance the opportunity to reduce synchronization and communication. One way is postponing computation in order to remove or weaken synchronization. The other is relocating computation which moves part or all of the computation done by a process in a critical section to anther process that does the computation better.

Issues of reducing synchronization and communication in parallel shared-memory programs have long been addressed by parallelization compilers via data dependency analysis. How much optimization they can reach therefore is affected by the accuracy of the static analysis. Rx approach, however, is based on run-time information in order to get precise conflicting-access records. To prescribe the program with the Rx approach, information collected during run-time must suffice for three types of operations: correlate line numbers of specific accesses with the source program, determine if a read uses values written by an earlier write, and determine the source operands of each write to memory. In addition, measurement overhead and performance perturbation should be controlled. For these ends, Rx uses a two-step process. First, it instruments the source program to gather certain information at run-time. This information is stored as state associated with the memory locations of the program. Then, when the program executes, Rx processes this state using a set of run-time algorithms, producing the information required for the analysis.

It is worth noting that a lot of strategies employed in Rx to reduce synchronization and communication are motivated by the observation of a large number of shared-memory applications. The designer of the tool identified many commonly-occurring scenarios of unnecessary synchronization in these applications, studied the characteristics of conflicting-access to memory in these scenarios, and figured out a transformation, without violating the semantics of sequential consistency model, of part or all computation involved in the scenarios in order to remove or weaken the synchronization. In contrast to other performance debugging tools that detect the presence of performance problem by quantitatively evaluating performance properties or metrics of concern, Rx approach looks for performance problem via qualitatively examining each occurrence of important programming constructs such as barriers, critical sections, flags, etc. This approach therefore can not only locate performance problems, but present possible transformation to correct the problems. On the other hand, performance optimization Rx approach can achieve is limited for two reasons. First, Rx looks at fundamental synchronization and communication pro-

gramming constructs with restricted semantics. Without relaxing the strict semantics of the constructs or using information about the semantics of the whole computation (rather than individual constructs) involved in the synchronization or communication, exploring better optimization is difficult. While Rx attempts to infer the intentions of the programmer from the programming constructs, the information most often comes from the programmers themselves[24]. Second, there are many other aspects, such as cache-relevant program behavior, load balance, etc., which have significant impact on performance, but are not addressed in the approach.

Kappa-Pi [25] is also a knowledge-based automatic performance analysis tool. In this tool, knowledge about commonly-seen performance problems is encoded into deduction rules. Rules are divided into different levels. The deduction process applies all rules in the first level to the trace events until no more facts are deduced. Then, the newly deduced facts serve as input to the next level of rules and the deduction process applies again. (It is not explicitly stated how these levels are demarcated though). The lowest-level rules involve primitive events like communication sends, receives of some processes, the highest-level rules may deduce some global collaboration schemes of the application like the master/worker, the rules in between bridge semantic gap between the two ends. In this way, higher-order facts can be deduced from lower-level events. The tool explains the problem found to the user by building an expression of the highest level deduced fact that includes the situation found, the importance of such a problem, and the program elements involved in the problem. The creation of a recommendation to repair the problem, however, is not aways feasible. It often requires more specific information about the program which allows the evaluation of some possibilities of changes in the source code. Kappa-Pi provides recommendation only when acquiring the information is possible.

One distinguishing feature of Kappa-Pi is that it tries to present better performance analysis hints to users by detecting higher level programming models. From the point of view of the programming constructions, the performance limits of an application are closer to user understanding and easier to explain. The tool, however, only touches upon two types of program constructions, Master/Worker and SPMD, focusing on detection of the constructions and of performance problems with respect to interplay of the construction with the underlying parallel machine. Kappa-Pi's knowledge base collects a very limited set of

performance problems and rules. It neither shows how to introduce new problem detection rules into the knowledge base, nor supports queries at different levels of abstraction. While Kappa-Pi introduces the possibility of using user-level information about program structure to analyze performance, we realize the possibility and propose a systematic approach to extracting knowledge from high level programming models.

Malony et al. [26, 27] point out that lack of a general theory of performance diagnosis is one of the main reason that performance diagnosis systems are not extensively used. They claim that a heuristic classification (HC) model of problem-solving is a sound basis for a theory of performance diagnosis but ultimately should be extended by model-based strategies. Elements of HC fitting to performance diagnosis include:

1. *solution space* that is composed of a set of predefined hypotheses that explain observed performance behavior;
2. *heuristic match* that matches features of the program's performance to hypothesis;
3. *abstraction* that extracts relevant features from experiment space (raw data);
4. *refinement* that relate generic hypotheses to program-specific explanation of behavior;
5. *strategy* that a PDS uses to specify the way that the three processes – (*heuristic match*, *abstraction*, and *refinement*) are interleaved and ordered.

They apply the HC model to explain many features of existing performance diagnosis tools, such as Paradyn, AIMS, and MTOOL. Based on the observation that the existing systems suffer from poor combination of automation and adaptability, they built an automatic performance diagnosis architecture, called *Poirot*. Poirot distinguishes itself from other performance diagnosis systems with two salient components – a problem-solver that assembles and runs diagnosis methods guided by user policies, and an environment interface that provides portable connection to supporting tools for performance data collection, analysis and presentation. The problem solver stores a variety of performance diagnosis methods and the associated rationales for method selection in a knowledge base. There is an reasoning *engine* through which Poirot chooses methods from the knowledge base and

executes them. The methods stored in the knowledge base take the form of *goal-action* rules, where a goal stands for a particular diagnosis task, and the corresponding action(s) represents the diagnosis action for accomplishing the task. The actions often need conducting experiment and collecting performance data. In this case, they send commands to tools via the environment interface, which then transforms the commands into primitive diagnosis actions that are executable by the tools.

The advantages of Poirot include: 1. Various existing diagnosis methods converge under a performance diagnosis theory, which makes comparison and evaluation of the methods possible. 2. Poirot separates diagnosis methods from the softwares that support the methods, thus supporting adaptable diagnosis. On the other hand, the heuristic classification model is limited in several aspects. First, HC assumes that program component structure is given. The assumption is reflected in the performance hypotheses, which typically state that a component (e.g., a routine) has a particular class of performance problem. In practice, it is not rarely seen that program components are unknown to the user until performance data is collected. Besides, decomposing program into components helps localize performance inefficiency, but it loses the context information that helps explain the inefficiency. Second, HC model assumes that all performance problems are given. However, in some existing performance diagnosis systems, unanticipated problems are sometimes identified. Third, Poirot presents a diagnosis theory from an architectural point of view that emphasizes adaptability and automation of performance diagnosis. The model barely addresses how to explain performance problems detected under the theory. The explanation is necessary for revising the program for better performance. In addition, there are also some other properties of equal importance to performance diagnosis, such as quality and efficiency. Determinant factors to these properties are, using terminologies in the heuristic classification model, definition of performance hypotheses, selection of the taxonomy of performance problems, and selection of heuristic match, abstraction, and refinement strategies. The performance diagnosis theory is insufficient in comparing and evaluating diagnosis systems as far as these properties are concerned.

A summary of the available performance diagnosis approaches is presented in the Table 2.1. In contrast to the tools and methods outlined above, our goal is very different. Our focus is on automatic performance problem discovery and explanation at a high level of

TABLE 2.1: Synthesis of parallel performance diagnosis techniques

System	Bottleneck-search	Explanation level	Automated	Diagnostic Strategy
Paradyn	Metric-based	low	yes	W^3 model
Expert, AKSUM	Property-based	low	semi-auto	Refine code regions and experiment env.
Cause-effect	Metric-based	medium	unknown	Waiting time analysis, compare process behaviors
ATExpert	Metric-based	high	yes	Refine code regions, use expertise of loop analysis
POETRIES	Metric-based	high	unknown	Model-based (only Master-worker case provided)
Rx	Metric-based	low	yes	Synchronization and communication analysis in shared-memory programs
Kappa-Pi	Metric-based	high	semi-auto	Model-based (only Master-worker and SPMD case provided)
Poirot	Metric-based	unknown	yes	Converge existing diagnosis methods

program abstraction. We intend to explain performance with program-sepcific behaviors and to support the causal reasoning in an automated manner.

Our approach also differs from the others on a number of points. We define application-specific, semantic-level metrics so as to enhance the explanation power of metric-based performance debugging. Our approach extends construct-pattern based performance interpretation to program-pattern based interpretation, and migrates the qualitative debugging methods from explicitly parallel shared memory programs to parallel and distributed programs. We use a bottom-up diagnosis strategy that promotes abstraction level of performance inferencing gradually by refining focus on program-specific behaviors.

2.5 Chapter Summary

In this chapter, we presented a new approach to performance diagnosis. Our approach incorporates performance knowledge into generic iterative diagnosis processes to address

the automation of performance diagnosis at a high level abstraction. Required performance knowledge is classified into four categories: experiment design and management, performance models, performance evaluation metrics, and performance factors at a high level of abstraction. Parallel computational models are examined and evaluated as a dependable source of performance diagnosis knowledge. At last we compare and contrast our approach with related work in the literature. In the next chapter, we will present our model-based performance knowledge engineering approach.

CHAPTER III

MODEL-BASED PERFORMANCE KNOWLEDGE ENGINEERING

A performance diagnosis tool can benefit from knowing the computational model, i.e., computational and communication pattern, of a parallel program. In the previous chapter, we presented potential advantages of parallel models to performance diagnosis and the performance knowledge categories. These include experiment design and specification, performance models, performance evaluation metrics, and performance factors at a high level of abstraction. we need to extract from the models to support automatic diagnosis processes. This chapter will show our methods for realizing the potential advantages of parallel models. We first present a systematic approach for extracting the expert knowledge required for performance diagnosis from parallel models and representing the knowledge in a manner such that the diagnosis process can be automated. We then demonstrate the effectiveness of our knowledge engineering approach through a case study of the parallel Divide-and-Conquer model.

3.1 Performance Knowledge Generation Based on Models

Extracting performance knowledge from parallel models involves four modeling stages, behavioral modeling, performance modeling, metric-definition, and inference modeling. The approach is shown in Figure 3.1. First we describe model behaviors in the format of abstract events, then form performance models based on the structural information in the abstract events. Semantics-aware performance metrics are formulated next to evaluate per-

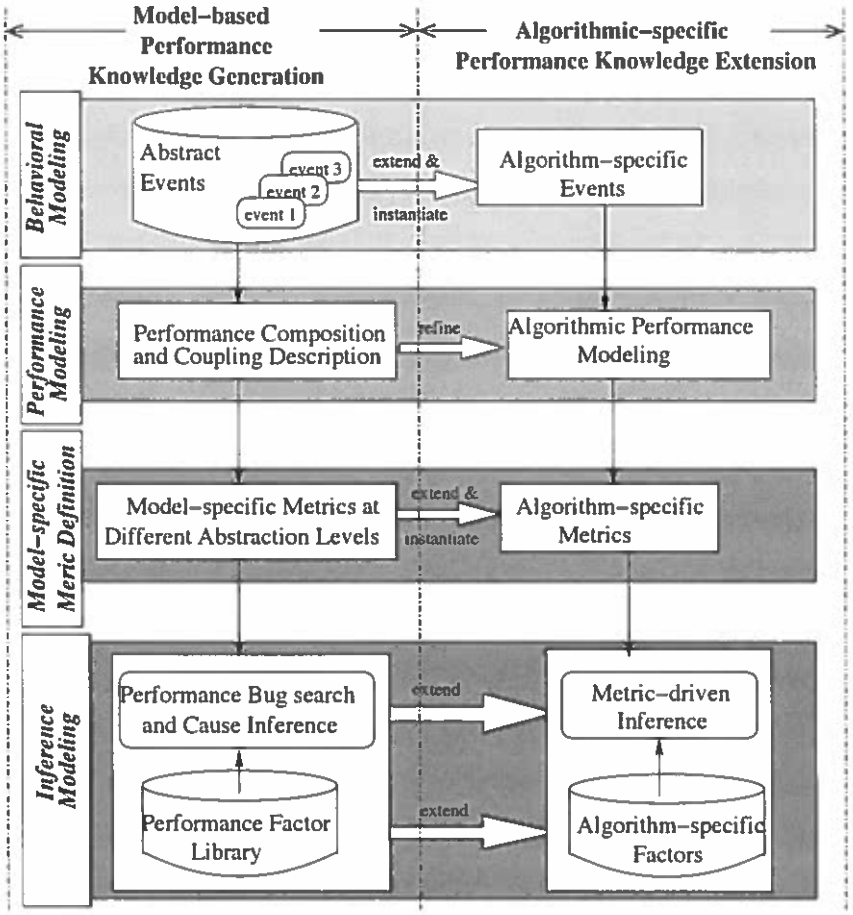


FIGURE 3.1: Generating performance knowledge from parallel models. Algorithm variants can derive performance knowledge from the basic generic model.

formance overhead types derived from the performance models. And finally the iteratively refined inference steps that use the products in the first three stages are formally captured with a tree structure.

Our four-stage knowledge engineering approach emphasizes the adaptability to model variants. We aim to support performance knowledge generation at two levels of program abstractions, model-based and algorithm-specific knowledge. Algorithmic implementations of a computational model may introduce new performance knowledge with regard to program behaviors, performance properties, performance-critical design factors, or cause inference. Following our four-stage knowledge extraction approach, the new knowledge can be generated by the users in the form of refinements or extensions of the generic model

knowledge, as shown in right hand part of Figure 3.1. Algorithm-specific knowledge generation can follow the same generic model-based knowledge extraction approach. In each stage of the knowledge modeling, we allow for the expression of algorithm variants that may add load-balancing, task scheduling, or other performance enhancements to a model implementation. And our knowledge representation form, i.e., inference trees, can be readily extended to incorporate the new knowledge into the inference system that is initially based on generic model knowledge.

3.1.1 Behavioral Modeling

Behavioral modeling captures knowledge of program execution semantics as behavioral models represented by a set of abstract event types at varying detail levels, depending on the complexity of the model and diagnosis needs. The purpose of the abstract events in the diagnosis system is to guide experiment design and specification, and to give contextual informaton for performance modeling, metric definition, and diagnostic inferencing.

As an instrumented program executes, it generates performance information based on event occurrence. Performance data associated with basic events lacks the context of the model-specific behavior. In the Master-Worker model, for instance, we cannot find a problem knowing only that a worker spends 2 seconds in a **MPI_Recv** routine call, while we may identify the master as a "late sender" by comparing entering time of corresponding **MPI_Send** at the master with the entering time of the **MPI_Recv**. Further causes of why a send was late may be found by looking at what the master was doing before entering the **MPI_Send**. To associate primitive performance information in terms of model-specific program behaviors, we introduce abstract events that consist of a set of related lower-level events and represent performance characteristics associated with the event interactions.

We adapt the *behavioral model description* used in EBBA [28] into Abstract Event Type Description (AETD). The AETD description of each abstract event type consists of one required component, expression, and four optional components, constituent event format, associated events, constraints, and performance attributes.

- *Expression.* An abstract event usually represents a sequence of constituent events. A constituent event can be a primitive event presenting an occurrence of a predefined

action in the program (e.g., inter-process communication or regular routine invocations), or an instance of some other abstract event type involved. The *expression* is a regular expression-like specification that names the constituent events and enforces their occurrence order using event operators. The order can be *sequential* (∘), *choice* (|), *concurrent* (Δ), *repetition* (+ or *), and *occur zero or one time* ([]).

- *Constituent event format* specifies the format and/or types of the constituent events. For primitive events, the format often takes the form of an ordered tuple that consists of the event identifier, the timestamp when the event occurred, the event location, etc. For constituent abstract events, their types are specified.

- *Associated events* are a list of related abstract event types, such as a matching event on a collaborating process or the successive event on the same process. The purpose of associated events is to formulate performance attributes that may involve multiple relevant abstract event types.

- *Constraints* indicate what attribute values an instance of an abstract event type must possess to match its corresponding expression members and associated events. Pieces of constituent events are recognized in accordance with the constraints as an event trace stream is scanned through. Different programs using a same computational model distinguish themselves from the others by the specifications of the constraints, which are often determined by their implementations.

- *Performance attributes* present performance properties associated with the abstract event type and their computing rules. The computing rules for evaluating the attributes will be filled in as a product of performance modeling. A performance attribute value will be calculated with the rules as the abstract event is instantiated with performance data. The value quantifies performance inefficiency due to an occurrence of the program behavior the abstract event represents.

We partition program behaviors into a set of abstract event types, each of which represents either system activities in a distinct computation phase, an interaction pattern of parallel components, or a segment of algorithmic solution. The abstract events can be at

varying detail levels depending on the performance inference needs. In the early inference steps, a coarse description of overall program behavior is sufficient. Having detected performance degradation happening with an abstract event type, we may zoom in to elaborate on its constituent event. We refine the definitions of constitute events with algorithm details, and use the refined behavior descriptions as a base for the next iteration of performance modeling and inferencing.

One salient advantage of abstract events is that they are able to describe dynamically changing program behavior and associated performance properties. Processes may change execution path with time due to dynamic control flow or have varying communication partners as data dependency changes. The dynamic attributes of parallel programs makes it difficult to locate and explain performance bugs. Behavioral model subsets are foreseeable from the algorithm, however, no matter whether occurrences of their instances are statically predictable or not. We capture occurrences of the behavioral models from program execution trace and identify performance properties associated with them, thereby enabling diagnose of performance losses due to dynamic program behaviors.

The abstract event descriptions can also be used to decide which specific performance experiments to conduct to collect the information most relevant to the state of problem investigation. The structural information helps captures only performance events that have direct relevance to hypotheses at the abstraction level the inference has reached so far. Incremental measurement data generation can effectively constrain the volume of performance information we have to consider.

Algorithm-specific behaviors can be expressed by extending already available model-based abstract event types. An algorithm using a design model often introduce new activities due to the specific problem solution such as load-balancing, task scheduling, etc. The new activities can also be described in abstract event format by inheriting generic model descriptions and adding in algorithmic extensions or refinements. In the next section we will show examples of abstract events for the D&C model and an algorithm implementation of D&C .

3.1.2 Performance Modeling and Metric Formulating

Performance modeling is carried out based on structural information in the AETDs. The modeling identifies performance attributes with respect to the behavior models represented by the abstract events and model-specific performance overhead categories.

In our methodology, a performance model is not a closed-form mathematical formula of system/application parameters. Rather we present descriptive performance compositions that consist of computational components/overhead categories. Given a computational model, participating processes can be grouped into distinct clusters (e.g., masters and workers in Master-worker model) in terms of their activities and their interaction modes with other processes. We generate a distinct performance model for each process cluster. Performance models defined in this way serve two goals: (a) performance models of individual process clusters focus on computational components specific to the cluster activities, which are responsible for possible performance losses happening in the cluster. (b) Differences in the performance models of inter-dependent clusters reflect their behavioral difference, therefore are useful for interpreting performance losses at interacting points such as communication and synchronization.

Performance attributes associated with each abstract event type are identified as a product of performance modeling. Performance attributes represent different performance overhead categories, which could be performance contribution of a computational component, or a pattern of performance inefficiency due to cluster interactions.

Performance metrics for evaluating the overhead categories are then defined, in terms of performance attributes in related abstract events. Performance attributes represent performance characteristics of one single abstract event type, while metrics sort out performance attributes in different abstract events that share the same semantics into one overhead category and define the rules for synthesizing the related events to evaluate the overhead. Performance evaluations with the model-specific metrics maintain the relevance to program context, which therefore makes it easier to explain the evaluations with model semantics.

Extending algorithmic-specific metrics from the generic model-based ones follows the same development path. Performance modeling based on the extended algorithm behaviors will produce algorithm-specific performance attributes. The new attributes join in the

overhead classification and are incorporated into metric formulations to reflect algorithm-specific semantics.

3.1.3 Inference Modeling

Inference modeling captures and represents the performance problem search and interpretation process formally. Refining problem search space is the most important step in the inference process in that it determines the direction of performance cause search and essentially boosts the abstraction level of inference. Our approach refines search space by first refining performance models to restrict attention to more specific performance aspects, then defining model-specific metrics addressing the performance aspects, and evaluating the metrics. Approaches to refining performance modeling therefore play a critical role in the refinement of search space. We identify following directions and methods for refining focus of performance examination, as shown in Figure 3.2:

(M1) *breadth decomposition* – decomposing performance cost according to computational components of a model and elaborating on each component;

(M2) *phase localization* – restricting to model-specific computational phases to look for performance losses occurring in the time periods;

(M3) *concurrency coupling* – focusing on and formulating performance coupling among interacting processes which arises from concurrency, workload distribution, data/task dependency, etc.;

(M4) *parallelism overhead formulating* – identifying and formulating parallelism-specific performance overhead due to, for instance, task scheduling, workload migration.

We will illustrate an application of these approaches in the case study section.

In our diagnosis approach, we aim to find performance causes (i.e., an interpretation of performance symptoms) at the level of the design of the parallel program, that is, performance-critical design factors specific to the particular parallel model. The high-level performance factors (e.g., in Master-Worker model, number of workers and load-balancing

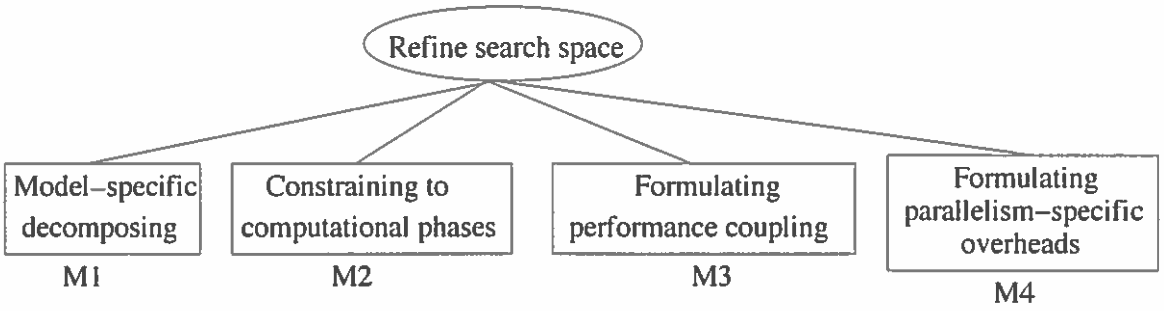
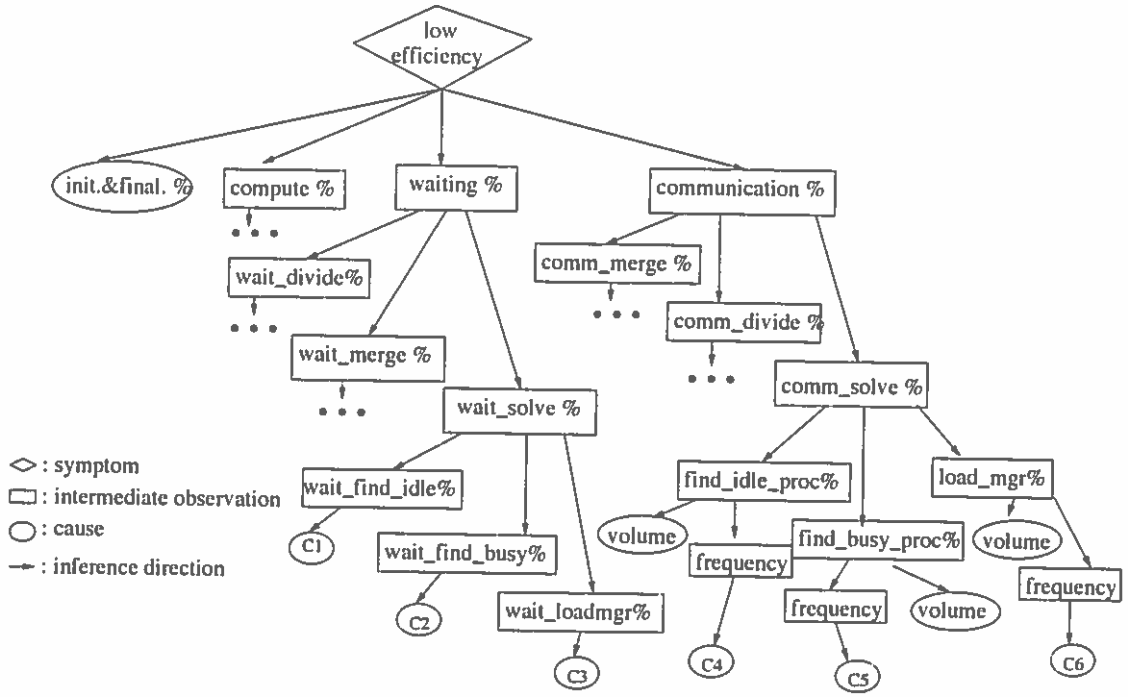


FIGURE 3.2: Approaches to refining search space.

method the master uses to assign jobs) can be collected from performance expertises of expert programmers, and will form candidate causes for interpreting performance problems.

Causal inference is the mapping of low-level performance measurement data to high-level performance factors to explain a performance symptom (i.e., an anomaly deviating from the expected performance, like low speedup, high parallelism cost, etc.). The bottom-up inference process is captured in the form of an inference tree where the root is the symptom to be diagnosed, the branch nodes are intermediate observations obtained so far and needing further performance evidences to explain, and the leaf nodes are explanations of the root symptom in terms of high-level performance factors. An inference tree for diagnosing the symptom "low efficiency" in D&C model is presented in Figure 3.3. An intermediate observation is obtained by evaluating a model-specific performance metric against the expected value or a certain pre-set threshold that defines the tolerance of severity of the performance overhead the metric represents. In Figure 3.3, for example, node *comm_solve* means the communication cost in the stage of solving problem cases. If it turns out to be significant comparing to the expected, the inference engine will continue to search for the node's child branches. The leaf nodes finally reached together compose an explanation of the root symptom. It is clear that inference processes presented in the trees are driven by metric evaluation. Performance knowledge associated with the metrics, including related abstract events, performance overhead types, and metric computing rules, are recalled only when needed by current inference step. Inference trees, therefore, formalize a structured knowledge invocation process. In addition, inference trees can readily incorporate knowledge generated from new experience, further performance model refine-



c1: Waiting time for finding a idle process is significant. It is very likely that the load–balancing algorithm is not efficient in locating idle processes.

c2: Waiting time for finding a busy process to migrate its workload is significant. It's likely that the load–balancing algorithm is not efficient in locating busy processes.

c3: Waiting time for the arrival of migrated workload is significant. The possible cause is that deciding workload to be migrated is expensive at the busy processes.

c4: There is a number of communications for finding idle processes to share workload. Change the load–balancing algorithm so that busy processes request help at a moderate frequency.

c5: There is a number of communication for finding busy processes to get workload. One possible cause is that the average workload per migration involves relatively small amount of computation time so that the process requests for extra workload pretty often. Another possible way to improve performance is to change the load–balacing algorithm so that idle processes request extra workload at a moderate frequency.

c6: Load migration is frequent. One possible cause is that the average workload per migration involves relatively small amount of computation time so that the processes request for extrat workload pretty often. Another possible cause is that some load migration trafers too much workload to an idle process so that orignial busy process becomes idle soon and the workload then thrashes between the two processes, causing unnecessary data transfer.

FIGURE 3.3: An inference tree for performance diagnosis of Divide-and-Conquer model.

ment, or algorithm-specific inference steps through adding branches at appropriate tree levels, making knowledge representation highly extensible.

The inference tree structure also has implications for experiment design. Since each node in the inference tree is associated with a distinct metric evaluation, the abstract event types relevant to the metric are retrieved to decide experiment specifications. Constraining clauses of the abstract events implicitly indicate program segments or routines to be instrumented. In addition, constituent event attributes specify performance data and type the experiment needs to record. Thus each experiment instance captures only performance events that have direct relevance to hypotheses the inference has reached so far. Performance data associated with branches of the inference tree that we never visited for some diagnosis cases are avoided.

In the next section, we will demonstrate how to generate performance knowledge from an example parallel model, Divide-and-conquer (D&C), using the approach presented above.

3.2 Case Study – Divide-and-Conquer Model Knowledge Generation

3.2.1 Model Description

The data-parallel Divide-and-Conquer (D&C) model describes a class of parallel programs that features recursively splitting a large problem into a certain number of smaller subproblems of the same type, then solving the subproblems in parallel and merging their solutions to achieve the solution to the original problem [8]. In this model, data are initially distributed over all participating processes. To split the work among the processes, all processes first sort out data locally, then exchange data in a specified order. The data redistribution results in two independent process sets whose data are disjoint (this is due to the nature of divide-and-conquer problems). Each set of processes then continues the process splitting until singular process sets are reached. Thereafter each process can independently work on its assigned data with serial divide-and-conquer code. After the independent computing has finished, processes merge their partial results with sibling processes' to form an

overall solution. The model behaviors are illustrated in Figure 3.4. Example problems that can be parallelized with this model include Quicksort, Barnes-Hut *n*-body [29], Delaunay triangulation [30], etc.

It is well-known that the D&C computing tends to be irregular and data-dependent. Often associated with the parallel D&C model is a load-balancing method that strategically migrates workload from overloaded processes to idle ones at runtime. The migration is illustrated in Figure 3.5. In the figure, the local serial D&C computation at individual processes is represented as a tree where the root of the tree represents the whole problem to be solved by the process, each branch node in the tree corresponds to a problem instance, and children of the node correspond to its divided subproblems. Each leaf represents a base problem instance where problem split stops. At runtime, process 3 migrates workloads represented by branch node 3 and 4 to process 4 and 2 respectively since these two processes have completed computing and are being idle. Without loss of generality, we make an assumptions about the load-balancing method that a load migration occurs only when there are idle processors available for doing extra computation.

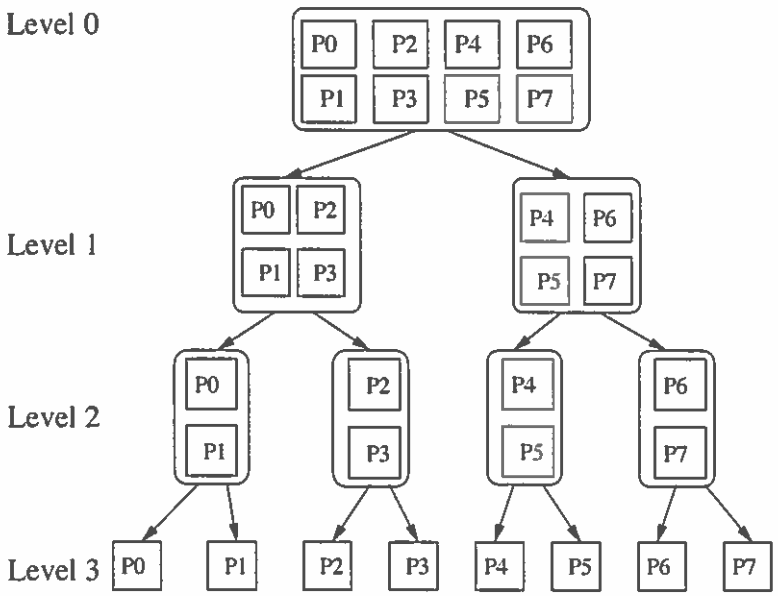


FIGURE 3.4: A D&C computation with 8 processors and three levels of problem splitting, where each splitting divides processors into two groups which work on orthogonal data sets independently. The processors merge sub-results with brother processes as designated by the splittings.

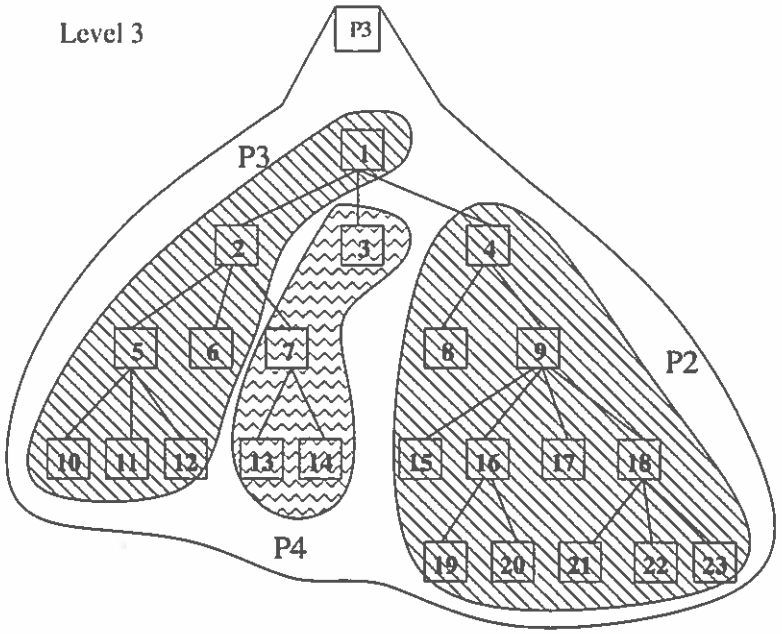


FIGURE 3.5: An illustration of load-balancing in D&C. P3 migrates workloads associated with node 3 and 4 to idle P4 and P2 respectively. Each square node represents a problem/data instance to be solved by a processor. It may divide into a set of child nodes of smaller problem sizes.

3.2.2 Behavioral Modeling

We model the D&C model behavior with abstract events depicted in Figure 3.6. The figure shows a level I event that sketches overall behavior of D&C, and level II events that refine the descriptions of essential model components. We display only the full description of event *Solve* in the figure for brevity. The performance attributes are derived from inter-process interactions, which will be used later for synthesizing model-specific metrics. Other fields of the event description, including constituent event format, constraints, and computing rules of performance attributes are filled in when a concrete algorithmic implementation of the model is generated, which we will illustrate with an example parallel application, Quicksort, next in section 5.1.

The program instrumented according to the abstract event specifications will generate only performance data relevant to the investigation of the specified program behaviors. Event instances matching the event specifications will be recognized from the performance data stream, and are used for calculation of model-specific metrics in later inference steps.

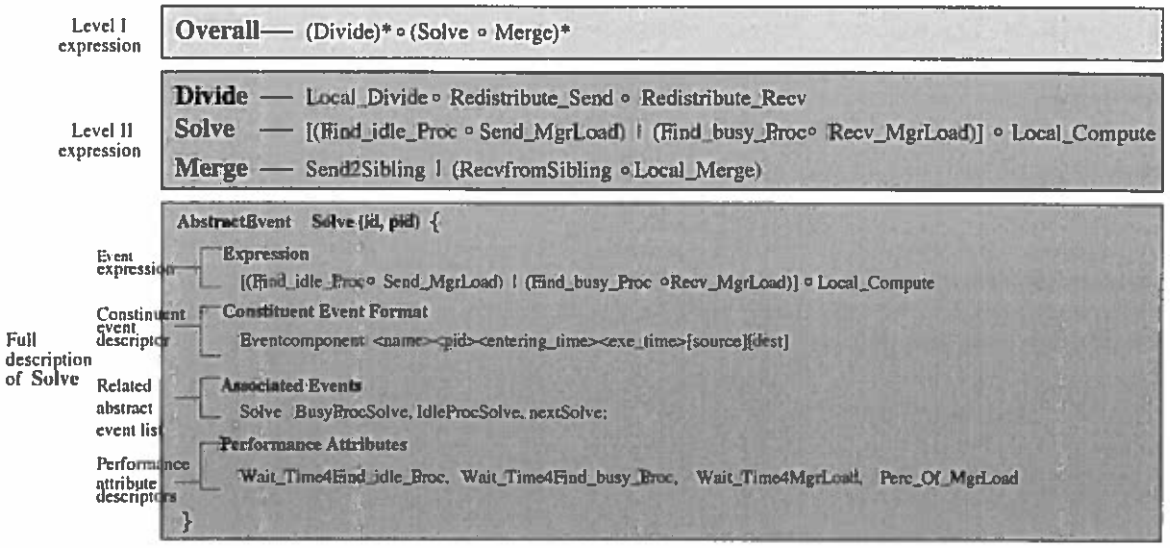


FIGURE 3.6: Two-level D&C abstract event descriptions.

3.2.3 Performance Modeling

Next we model D&C performance by referring to the structural descriptions in the abstract events. Following phase localization rule in section 3.1.3, total elapsed time of a processor p in a parallel D&C execution, denoted as t_p, consists of t_{init} (process initialization cost), t_{comp} (computation time), t_{comm} (communication cost for transferring data among processors and synchronization cost), t_{wait} (amount of time spent waiting for data transfer or synchronizing with other processors), and t_{final} (process finalization cost):

$$(\mathrm{M2}) \Rightarrow t_p = t_{init} + t_{comp} + t_{comm} + t_{wait} + t_{final} \tag{3.1}$$

Whenever we refer to communication time in the paper, we mean effective message passing time that excludes the time loss due to communication inefficiencies such as late sender or late receiver in MPI applications. Rather, waiting time accounts for the communication inefficiencies with the purpose of making explicit performance losses attributed to mistimed processor concurrency, hence parallelism design.

According to the level I event and breadth decomposition rule, computation time, communication time, and waiting time can be categorized into three classes, time spent in splitting problems, solving problem cases, and merging sub-solutions.

$$(\mathrm{M1}) \Rightarrow t_{comp} = t_{divide} + t_{solve} + t_{merge} \tag{3.2}$$

$$(\mathrm{M1}) \Rightarrow t_{comm} = t_{comm-divide} + t_{comm-solve} + t_{comm-merge} \tag{3.3}$$

$$(\mathrm{M1}) \Rightarrow t_{wait} = t_{w-divide} + t_{w-solve} + t_{w-merge} \tag{3.4}$$

Level II events dictate refined performance models. The time spent in problem solving, for instance, is categorized into three classes, finding an idle process to share workload, finding a busy process to obtain extra workload, and migrating workload, according to the abstract event *Solve* defined in Figure 3.6.

$$(\mathrm{M1}) \Rightarrow t_{solve} = t_{local\_comp} + (t_{comp-find\_idle\_proc} \; or \; t_{comp-find\_busy\_proc}) \tag{3.5}$$

$$\begin{aligned} (\mathrm{M1}) \Rightarrow t_{comm-solve} \quad &= \quad t_{comm-find\_idle\_proc} + t_{comm-send\_mgr\_load} \\ &or \quad t_{comm-find\_busy\_proc} + t_{comm-recv\_mgr\_load} \end{aligned} \tag{3.6}$$

Performance coupling of a busy processor with a idle processor in a load migration event induces possible waiting time at interaction points of the two processors:

$$(\mathrm{M3}) \Rightarrow t_{w-solve} = t_{w-find\_idle\_proc} \; or \; (t_{w-find\_busy\_proc} + t_{w-mgr\_load}) \tag{3.7}$$

3.2.4 Model-specific Metric Definition

The performance models above enable the definition of model-specific metrics to use for performance evaluation. Model-specific metrics are calculated by categorizing and synthesizing performance attributes in related abstract events. Formulating each item in equation (3.7), for instance, we get a set of metrics that are associated with load-balancing

performance as below:

$$t_{w-find\_idle\_proc} := \sum_{i=0}^{K} e^{i}_{Wait\_Time4Find\_idle\_Proc} \tag{3.8}$$

$$t_{w-find\_busy\_proc} := \sum_{i=0}^{K} e^{i}_{Wait\_Time4Find\_busy\_Proc} \tag{3.9}$$

$$t_{w-mgr\_load} := \sum_{i=0}^{K} e^{i}_{Wait\_Time4MgrLoad} \tag{3.10}$$

where $e^{i}_{Wait\_Time4Find\_idle\_Proc}$, $e^{i}_{Wait\_Time4Find\_busy\_Proc}$, and $e^{i}_{Wait\_Time4MgrLoad}$ represent the performance attribute *Wait_Time4Find_idle_Proc*, *Wait_Time4Find_busy_Proc*, and *Wait_Time4MgrLoad* of the *i*th *Solve* event e^{i}occurring on the process respectively, and k the number of event instances *Solve*.

The performance modeling and metric definition in terms of model semantics provide a foundation to the generation of inference trees that formally represent the performance problem search and interpretation process.

3.2.5 Inference Modeling

For diagnosing D&C programs at a high level of abstraction, we need to identify design factors critical to parallel D&C performance. We will interpret performance problems in terms of these factors. In general, the following factors have the most impact on D&C performance:

- *Concurrency and cost of divide and merge stage.* Due to the nature of D&C problems, it is very likely that not all of the processes are active at divide or merge stage. In the case of low concurrency, if divide or merge operations are expensive, the idle processors will wait a significant amount of time for problem dividing or solution merging, therefore insufficiently utilized.

- *Size of base problem instance.* D&C model is particularly effective when the amount of work required for solving a base case is large compared to the amount of work

required for recursive splits and merges. On the other hand, for the purpose of maximizing processor utilization there should be a sufficient number of base problem instances. The two often conflicting conditions give rise to the trade-off between the depth of recursive split and the size of the base problems.

- *Scheduling algorithms used for balancing workload.* The algorithm decides where to migrate workloads and the amount of work to be migrated. The factor decides the degree of load balance and communication cost of moving workload around.

The generation of an inference tree for diagnosing D&C programs is based on performance modeling and model-specific metric evaluation. An example inference tree for diagnosing a symptom "low-efficiency" is presented in Figure 3.3. For brevity we present in the figure only inference steps that refer to the performance models and metrics we defined in section 3.2.3 and 3.2.4. We can see that inference trees represent a bottom-up performance cause inference approach that links low-level symptoms to causes at high level of abstraction. As inference process going deep, causes of performance inefficiency are localized.

The knowledge inference present in Figure 3.3 is not meant to be complete. A specific implementation of the model in an algorithm may introduce new behavioral models and performance factors. Nevertheless, designed to be extensible, our inference trees can readily accommodate the knowledge extension.

Another important thing to know about the inference tree is that nodes at different tree levels may enforce varying experiment specifications. Our diagnosis system can designate the experiments accordingly to collect performance data needed by the metric evaluation at an inference step.

3.3 Chapter Summary

This chapter described a systematic approach to generating and representing performance knowledge for the purpose of automatic performance diagnosis. The methodology

makes use of operation semantics and parallelism found in parallel computational models as a basis for performance bug search and explanation. In order to generate performance knowledge from computational models and apply it to diagnosing realistic parallel programs, we specifically identify methods for behavioral model representation, performance modeling, metric definition, and performance bug search and interpretation. The methods address not only performance cause interpretation at high-level program abstractions, but adaptivity to allow algorithm and implementation variants. We illustrated the knowledge generation approach with the Divide-and-Conquer model.

In the next two chapters, we will show the Hercule framework which offers a prototype automatic performance diagnosis system based on the extracted model knowledge. We will demonstrate the use of Hercule on three parallel applications that represent different computational models.

CHAPTER IV

HERCULE AUTOMATIC PERFORMANCE DIAGNOSIS SYSTEM

We introduced the model-based approach to generating performance knowledge that can support automatic diagnosis at a high level of abstraction in the previous chapter. In this chapter, we describe how to apply the knowledge in a real diagnosis system. We present Hercule, a prototype automatic performance diagnosis system, that implements the model-based performance diagnosis approach. We will discuss Hercule design issues, how to use Hercule from a user's perspective, and validation of Hercule diagnosis results.

4.1 Hercule Design

We have built a prototype automatic performance diagnosis system called *Hercule*[1], which implements the model-based performance diagnosis approach discussed above; see Figure 4.1. The Hercule system operates as an expert system within a parallel performance measurement and analysis toolkit, in this case, the TAU [4] performance system. Hercule includes a knowledge base composed of an abstract event library, metrics set, and performance factors for individual parallel models.

Given a program to be diagnosed, Hercule starts with information about the model the program uses, then comes up with experiment specifications in terms of abstract event descriptions of model behavior. The performance measurement toolkit integrated in Hercule

[1]The name was chosen in the spirit of our earlier performance diagnosis project, *Poirot* [27].

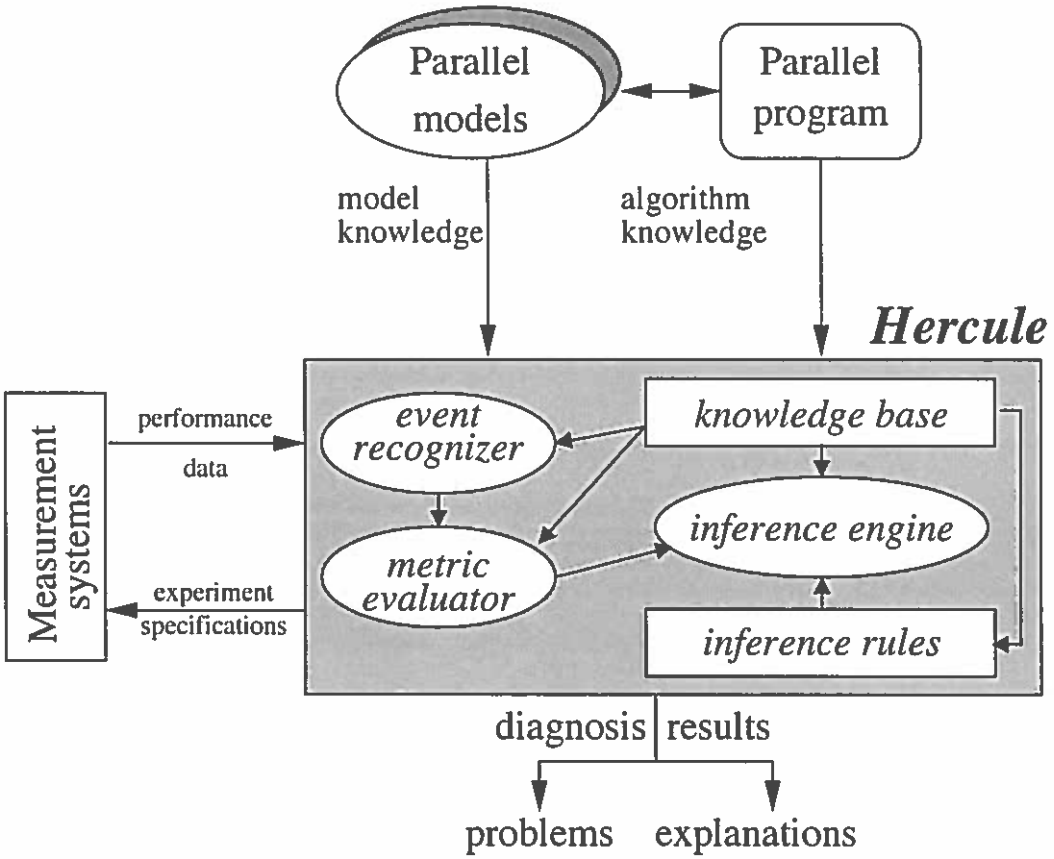


FIGURE 4.1: Hercule diagnosis framework

system conducts experiments according to the Hercule instructions and generates performance data specifically for the current inference step. The *event recognizer* in Hercule fits event instances into abstract event descriptions as performance data stream flows through it. Hercule then retrieves relevant metric formulation rules from the knowledge base to prepare for metric evaluation. The abstract event instances, along with the metric definitions are fed into Hercule's performance evaluator - *metric evaluator*, to generate performance metric values. The *inference engine* then takes in the metrics, evaluates the metrics, and decides on the next hypothesis to test. The engine actions are driven by encoded inference rules in Hercule. Hercule reaches diagnosis conclusions after iterations of experiments and analysis. These conclusions are output as the performance problems found in the program and corresponding explanations.

4.1.1 Encoding of Performance Knowledge

Hercule implements the abstract event representation in a Java class library. This library provides a general programmatic means to capture computational model behaviors. Each event type is implemented as an abstract Java class with a set of parameters that represent model components or constructs the event refers to. Given a program, the user provides names or values of the parameters used in the model implementation, thus producing an instantiated event class that is tailored to the specific program. The parameter values are also used to specify the *constraint* clauses in order to identify model components in the performance data stream. The constraints are implemented in the Java class as a method that examines whether a performance trace event satisfies the clause or not. The satisfying events will be instantiated constituent events in the description. Performance models associated with an abstract event type are coded as performance attributes and corresponding formulations, so they are encapsulated in the same Java class of the abstract event. The encoding of abstract events allows for algorithm extension due to model variants.

Model-specific metrics will be evaluated based on the related abstract event types. The metrics are represented in Hercule as a set of rules that classify and synthesize performance attributes in the relevant abstract events. Recall that the metrics are at different levels of program abstraction. *Communication time and computation time*, for instance, are generic metrics without reference to model semantics. While t_{w-mgr_load} in section 3.2.4 is specific to the D&C model behaviors. We couple high-level performance factors with the performance metrics at the highest level of program abstraction. Specifically, if a possible evaluation of a metric can be explained directly by a performance factor, we store the factor jointly with the formulation rules of the metric to make it easier to retrieve relevant knowledge in diagnosis process. Model-variant metrics not captured in the knowledge base are required to include the relevant performance factors to be incorporated into Hercule system.

Event recognizer and metric evaluator are coded in Java for the purpose of easily interfacing to the knowledge base. The event recognizer reads in from the knowledge base the abstract event types to be used in current inference step, and matches incoming performance data with the descriptions to generate valid abstract event instances. The instantiated abstract events are handed to the metric evaluator. The evaluator first retrieves relevant per-

formance metrics and then calculates the metrics using the event instance data. The event recognizer and metric evaluator can incorporate algorithm-specific abstract event definitions or metric computing rules written in Hercule format.

The effort involved in implementing performance knowledge base for a computational model consists of two parts: acquiring knowledge with the approach presented in the previous chapter and encoding the knowledge, including abstract event specification and performance metric formulation in Hercule-readable format. Work time needed for a performance analyst to generate knowledge varies depending on computational complexity of the model and desired detail level of the targeting inference tree. When using the knowledge base to diagnose a parallel application based on a parallel model, the user may need to express the programatic or algorithm variations with respect to abstract event descriptions, metric computing specifications, and corresponding inference tree. Because the generic knowledge base is inherited, additional efforts are reduced to adding knowledge specialization.

4.1.2 Encoding of Inference Steps and Hercule Inference Engine

Perhaps the most interesting part of the Hercule system is the cause inferencing system. The expert knowledge used to reason about performance problems based on model symptoms is structured as *inference trees* where the root is the symptom to be diagnosed, the branch nodes are intermediate observations obtained so far, and the leaf nodes are an explanation of the root symptom in terms of high-level performance factors. We transform the inference trees into production rules in Hercule system. Formally, a production rule is a condition-action statement in which the conditions match the current situation and the actions add to or modify that situation [31]. In performance diagnosis terms, a rule consists of one or more performance assertions, and performance evidences that must be satisfied to prove the assertions. Hercule makes use of syntax defined in the CLIPS [32] expert system building tool to describe production rules. The syntax takes the form:

```
(defrule <rule-name>
   <condition-element>*
   =>
   <action>*)
where
```

```
condition-element::= <pattern-ce> |
                     <assigned-pattern-ce> |
        <not-ce> | <and-ce> | <or-ce> |
              <logical-ce> | <test-ce> |
               <exists-ce> | <forall-ce>
action ::= <constant> | <variable> |
                  <function call>
```

Due to its extensibility, capabilities, and low-cost, CLIPS has been used in building expert systems of a wide range of applications in industry and academia. Hercule uses the inference engine provided in CLIPS to support automatic performance inference. The engine can repeatedly fire rules with original and derived performance information until no more new facts can be produced, thereby realizing automatic performance experiment specification and cause reasoning.

Inference trees already capture the main control structure and reasoning thread of performance inferencing, so we can immediately transform information embedded in the trees into production rules. Translated production rules are prioritized in terms of the performance problem searching and inferencing order as specified by the inference tree hierarchy.

Regular production rules test performance hypotheses using metric evaluations. We developed a CLIPS interface to the metric evaluator so that the inference engine can fetch performance metrics easily from the evaluator in the form of external function call. The interface also supports the passing of values/parameters between the engine and the evaluator. The benefit of the interfacing is that, other than addressing the whole program execution, performance metrics can be restricted to a subset of system behavior, such as a problematic processor, a degrading computation phase, or a set of semantically related program sections, with the parameter specifications. Thus inference engine can refine performance problem searching and focus on specific program behaviors.

For the purpose of automatic performance diagnosis with minimum user intervention, we need additional rules to support specification of experiments that generate necessary performance data. A performance diagnosis process includes deciding on what performance experiments to conduct to collect the information most relevant to the state of problem investigation. Thus, each experiment instance captures only performance events that have direct relevance to hypothesis at the abstraction level the inference has reached so

far. The incremental measurement data generation can effectively constrain the volume of performance information we have to consider. Performance data associated with branches of inference tree that we never visit for some diagnosis cases are avoided.

Our experiment specification indicates which inference tree level (i.e., abstraction level of the relevant metrics) the experiment serves, input problem and system parameters, and abstract events the experiment captures. Constraint clauses of the abstract events implicitly indicate program segments or routines to be instrumented. And constituent event attributes specify performance data and type the experiment needs to record. For example, in terms of the figure 3.6, if we detect that waiting time at *Solve* stage is significant with performance information from one experiment, then we can make another experiment collecting more specific data. Referring to the following production rule, the level 3 experiment is conducted with the same input problem and system parameters, but generates performance information associated with abstract event *Solve* to attempt to identify a higher-level factor.

```
(defrule do_experiment_level3
  (wait_solve_sig)
  =>
  (assert do_exp3)
  (assert (experiment
           (level 3)
           (input_problem SAME)
           (system_setting SAME)
           (abs_event_list Solve))))
```

4.2 Hercule Application from the Users Perspective

User involvement in using Hercule diagnosis system is twofold: specifying system parameters and extending performance knowledge to adapt to model variants. The users need to instantiate abstract event descriptions with their program information. *Constituent event format* is to be filled in with performance data format used by the measurement system, and *Constraint clauses* refer to program segments or routines that correspond to model

components in the *Expression* part. At present, Hercule generates performance experiment specifications in plain text form. The users need to conduct the experiment instructions on their targeting measurement system to collect performance data, and feed the data into Hercule to fuel performance inferencing. In diagnosing a program execution, the users can optionally provide thresholds for evaluating performance metrics (to assert if there is a problem associated with the metrics that is worth further investigation). In default, Hercule investigates every model-specific performance aspect, and provides explanations for the associated performance losses in a model execution. While with the thresholds, the users will make Hercule focus on the most concerned model properties or intentionally avoid some performance inference steps for their diagnosis cases.

We have discussed in the previous chapter that algorithmic implementation of a parallel model may introduce new performance knowledge with respect to behavioral models, performance properties, performance-critical design factors, and causal inferencing, as shown in Figure 3.1. The users can derive the new knowledge from the already built model-based knowledge using the engineering approach presented in the section 3.1 and integrate the knowledge into Hercule system to analyze their programs.

An algorithm may address model-specific data/control dependency in a particular manner or introduce extra management work to improve process utilization. The users describe the behavioral models distinct to the algorithmic implementation using algorithmic-specific abstract events. The new events can be refinement or extension of available model events. The increasing complexity of computation and process interactions will introduce more overhead categories and the need to refine performance model. Following the generic modeling approaches we presented in the section 3.1.2, algorithmic performance modeling can elicit potential performance overheads based on the structural information provided in the new abstract events, and lead to the formulation of algorithmic-specific metrics that evaluate the identified performance overheads. Algorithmic instantiation of a model may introduce new performance factors, which are possibly attributed to, for instance, the task scheduling strategy or data partitioning and mapping method used. To investigate performance effects of the new factors, they need to be added to the knowledge base along with the associated performance metric definitions. The new performance inferencing steps that refer to the algorithm-specific knowledge are to be incorporated into the origial model-

based inference tree in the form of branches at appropriate tree levels, and finally to be translated into production rules that are executable in Hercule system.

The effort involved in adding knowledge specialization of an algorithm varies depending on computational complexity of the algorithm and desired detail level of the targeting inference tree. Since the generic knowledge base is inherited, Hercule diagnosis can capture the set of basic performance problems in a model-based algorithm without any algorithmic specialization. On the other hand, algorithmic variants not addressed in the original model knowledge need to be encoded into Hercule system to conduct performance investigation at the desired level of details. The effort, however, will be amortized over many diagnosis cases of the same algorithm, and offset the otherwise laborious manual work required.

4.3 Validation of Hercule Diagnosis Results

In general, how is a performance diagnosis system to be validated? In order to test Hercule, we want a controlled way to run parallel programs, instrument the program according to the experiment specifications from the diagnosis system, and generate performance data in desired forms. However, in addition, a validation environment should also support the injection of known (model level) performance problems in a parallel program[2]. The diagnosis system is unaware (*a priori*) of the performance fault and thus sees the parallel system as a black box. Once the diagnosis process has completed, the validation environment can evaluate the goodness of the result with respect to the known problems introduced.

Figure 4.2 shows how Hercule and the validation environment work together. When a new parallel model is included in the system, we run test cases (either synthetic parallel programs or real-world applications) to evaluate Hercule diagnosis results. Given a program to be diagnosed, Hercule is only informed of the model the program uses. It reaches diagnosis conclusions after iterations of experiments and analysis. These conclusions are output as the performance problems found in the program and corresponding explanations. We then compare the conclusions against the performance problems introduced at the start.

[2] *Fault injection* is a common part of software testing and diagnosis environments.

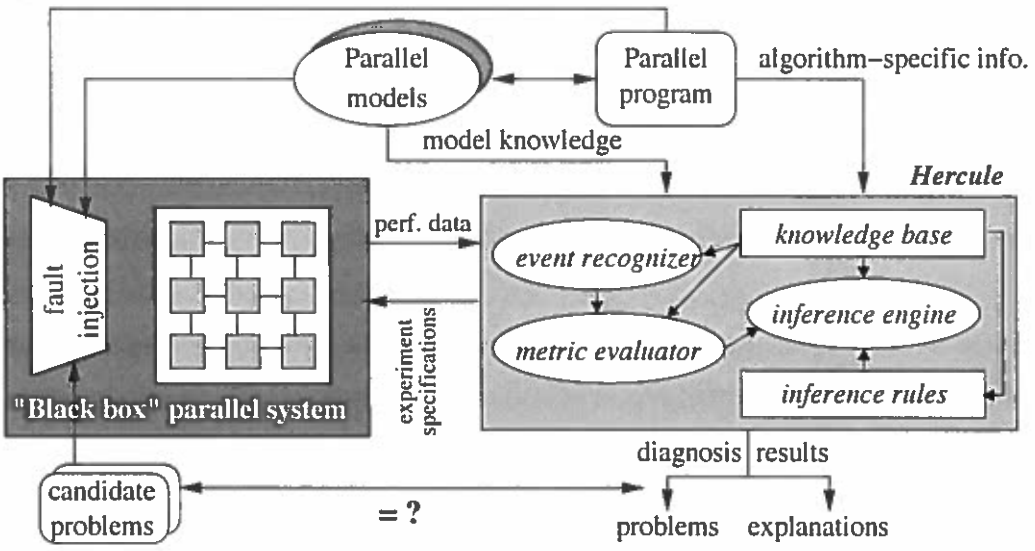


FIGURE 4.2: Hercule and performance diagnosis validation system.

Naturally there is another methodology for Hercule system validation, that is to fix performance problems in terms of Hercule conclusions and to see how well the tuning improves performance. This method naturally involves two aspects of evaluation: how fast the users can find where to fix and how well the performance is optimized. Since Hercule results interpret performance losses with model factors at a high level of abstraction, repairing the problematic factors most possibly leads to performance gains. How to tune the factors, however, often requires a greater degree of expertise about parallel problem-solving and understanding of the specific solution to the targeting problem. When there are more than one problematic design factor in a parallel program, for instance, tuning all of them at the same time may not generate the expected performance effects. A modification that optimizes performance in one test case may slow down execution in another case. The quality of the optimization phase is therefore hard to evaluate from a diagnosis tool developer's perspective. An interesting area for future work is to include in Hercule tuning-supportive facilities that helps design performance tuning trials in terms of diagnosis results and supports relative performance diagnosis that compares multiple executions with varied factor values and reasons about their performance differences.

4.4 Chapter Summary

In this chapter, we introduced Hercule - a prototype automatic performance diagnosis tool that implements our model-based performance diagnosis approach. The core of Hercule is a knowledge base composed of an abstract event library, metrics set, and performance factors for individual parallel models. Requiring only model-implementation information from the user, Hercule recognizes event instances matching the abstract event specifications as a performance data stream flows through it, calculates the associated performance attributes values, and synthesizes model-specific metrics for performance evaluation. Performance inference steps are encoded into production rule. The inference engine in Hercule is particularly helpful in performance diagnosis because it can repeatedly fire rules with original and derived performance information until no more new facts can be produced, thereby realizing automatic performance experiment generation and cause reasoning.

Hercule implementation mainly involves the encoding of model-based performance knowledge and inference processes, and interfacing knowledge base to inference engine to support automatic diagnosis. The implementation also addresses the adaptability to model-variants introduced by algorithmic implementations. A black box parallel experiment system is integrated into Hercule to validate diagnosis results. The validation system supports a controlled way to run experiments and the injection of known (model level) performance problems in a parallel program. It enables us to evaluate Hercule diagnosis conclusions on a sound ground. In the next section, we will present Hercule experiments with a set of well-known, widely applicable parallel applications.

CHAPTER V

HERCULE EXPERIMENTS

In the previous chapter, we introduced Hercule, a prototype automatic performance diagnosis system. We described how the Hercule framework implements the model-based performance diagnosis approach and applies the engineered performance knowledge in a real diagnosis system. In this chapter we will demonstrate Hercule's ability to diagnose performance problems with three parallel applications that represent D&C, Master-worker, and Wavefront model respectively. We will also investigate relative performance diagnosis, a specific application scenario of model-based approach, to demonstrate the effectiveness of our approach.

5.1 Divide-and-Conquer Model and Parallel Quicksort

Section 3.2 have presented performance knowledge engineering process of parallel Divide-and-Conquer (D&C) model. In this section, we demonstrate Hercule's ability to diagnose performance problems in a D&C program, parallel Quicksort. Experiments were run on a IBM pSeries 690 SMP cluster with 16 processors. The parallel Quicksort algorithm, using the aforementioned Divide-and-Conquer model, first recursively divides processes until the following state is attained – for any two processes P_i and P_j, if $i < j$ then any data element on P_i is less than or equal to any element on P_j. Then each process independently executes serial version Quicksort. There is no merge stage in the parallel Quicksort. Since a poor choice of pivot in Quicksort may lead to imbalanced data distri-

bution over processes at runtime, we employ a simple runtime load balancing system that uses a centralized manager to manage idle processes and to assign them to busy processes to share work load. Whenever a processor finishes local data sorting, it registers at the manager. Whenever a processor is about to recurse on more than a predefined amount of data in the serial phase, it requests the manager for help. If no idle processor is available, the manager sends back no-help response, and the original busy processor must continue computing on its own. Otherwise, the manager selects an idle processor, and tells it which processor to help and the amount of data to expect. The manager also informs the busy process where to migrate load. The busy processor then migrates the data to the helper, continues with its remaining data load, and waits for the helper to return the result. Figure 5.1 represents the actions and relationship of helper (idle) processes, helpee (busy) processes, and the manager as described in the abstract events *Solve, Req2MngEvt, and ManagerEvt.* The *Solve*

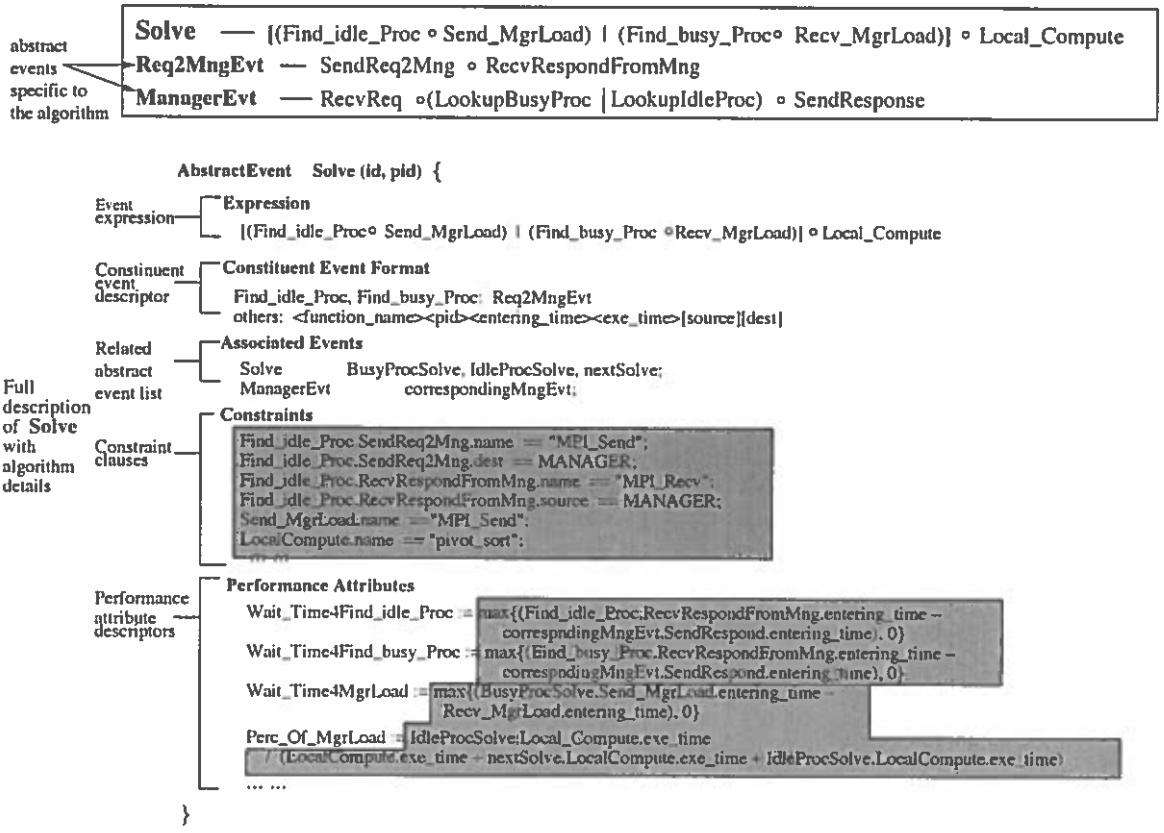


FIGURE 5.1: Extended abstract event descriptions of *Solve* in the parallel Quicksort algorithm. The shaded areas are instantiated by the Quicksort implementation of D&C model.

event description instantiated with the Quicksort algorithm is also shown here. Hercule recognizes event instances matching the event specifications as a performance data stream flows through it, calculates the associated performance attributes values, and synthesizes model-specific metrics for evaluation in later inference steps.

Table 5.1 presents major performance metrics we identify for D&C model.

In Figure 5.2, the Vampir timeline view of a *Solve* phase with load balancing in a Quicksort run with five processes is shown. It also depicts activity chart of the phase. The event trace is generated by the TAU [4] performance measurement system with only major model/algorithm components being instrumented. In the figure, dark red regions represent effective computation. Light yellow regions represent MPI function calls, including **MPI_Init**, **MPI_Send**, **MPI_Recv**, etc. Note that blocking/waiting time of processors is implicitly included in the elapsed time of blocked **MPI_Send** and **MPI_Recv** operations. In following performance diagnosis, we focus on investigating efficiency of the load-balancing aspect of the Quicksort.

TABLE 5.1: Performance Metrics of Quicksort.

	Name	Meaning
Model-specific Metrics	granularity	T_{comp}/T_{comm} with respect to D&C trees
	T_{wait_divide}	waiting time for exchanging data in splitting phase
	T_{wait_merge}	waiting time for merging results with sib processes
	T_{wait_solve}	waiting time in the solve stage
	$T_{wait\_find\_idle\_proc}$	(idle processes) waiting time for task assignments
	$T_{wait\_find\_busy\_proc}$	(busy processes) waiting time for getting a helper
	$T_{wait\_mgr\_load}$	(idle processes) waiting time for arrival of work assignment
	F_{mgr}	frequency of migrating data to other (idle) processes
	L_{mean_mrg}	average work load migrated to other (idle) processes

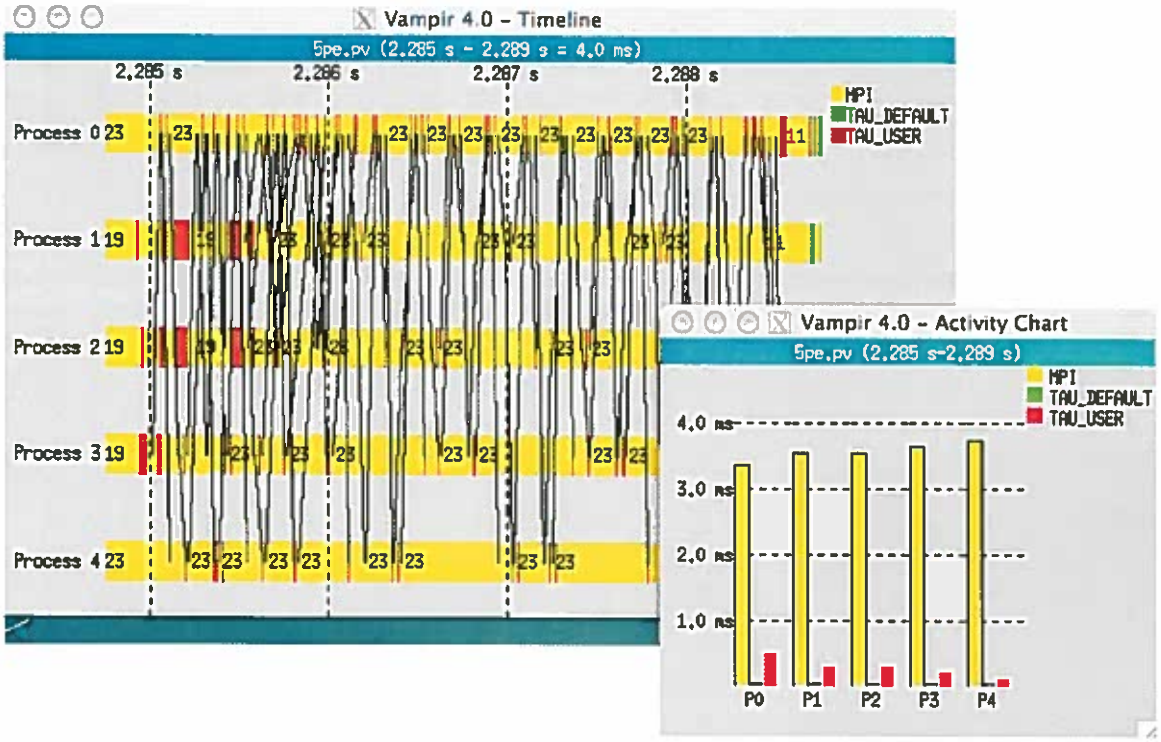


FIGURE 5.2: Vampir timeline view in a *Solve* phase of a parallel Quicksort run with five processes.

We can see that most of the time is spent in communication (i.e., light yellow areas). It is difficult to tell the communication pattern and what causes the communications in the event trace stream. However, our abstract event descriptions can fit the seemly intractable performance data into recognizable patterns and help infer performance causes. Hercule interpretations of *Solve* stage communication performance in this Quicksort run is presented below.

```
dyna6-221:~/PerfDiagnosis/bin lili$ testDC DC.clp 5pe.dup
Begin diagnosing DivideConquer program
... ...

-------------------------------------------------------------
Level 1 experiment -- collect performance data for computing
granularity associated with each D&C tree.

-------------------------------------------------------------
do experiment 1... ...

-------------------------------------------------------------
At Solve stage of D&C pattern, each process independently
```

runs sequential D&C code, which can be viewed as a process of expanding and collapsing a D&C tree. In the case of load-imbalance, some branches of the D&C tree on a process may be migrated to another process which has finished computing. We explain performance with respect to the D&C tree structure.

The D&C tree originally rooted at process 2 has as low computation/communication ratio as 0.019.

Level 2 experiment -- generate performance event trace for evaluating communication performance of Solve in D&C tree 2.

---

do experiment 2... ...

Among the communications with respect to tree 2, moving data to other processes to balance work load comprises 15.09% of overall communication time, and requesting the manager for finding a idle processor comprises 84.9%.

One possible factor contributing to the expensive requesting-manager communication cost is that busy processes send request to the manager to find a idle process at a high frequency. Users are advised to check the amount of work a process has to get done prior to requesting the manager to make sure that the processes request the manger at a moderate frequency.

Small size of base problem instance is also a possible cause of too much commnication with the manager because the communication frequency is proportional to recursion depth.

Level 3 experiment -- generate performance data for computing load migration caused performance overheads.

---

do experiment 3... ...

In average, every work load migration inappropriately transfers 89.8% of remaining work to idle processes, which causes load imbalance in the subsequent D&C tree computing, thus more communication due to load migration. In D&C tree 2,

```
work load migrated to other processes is partially migrated
back to the original process that initiates the tree,
causing workload thrashing and unnecessary data transfer.
============================================================
Diagnosing finished...
```

Recall that recursive process splitting in D&C model leads to a hierarchy of process sets. Each process independently runs sequential D&C code, which can be viewed as a process of expanding and collapsing a *D&C tree* [33] as shown in Figure 3.5. In the case of load-imbalance (i.e., D&C trees on different processes involve varied workload), some branches of the D&C tree on a process may be migrated to another process which has finished computing. Hercule evaluates and explains performance from the point of view of the D&C trees since the structure makes it easy to interpret inter-process communications for load migrations.

Given the program and performance knowledge associated with D&C pattern, Hercule automatically request an experiment that collects performance event trace for evaluating efficiencies (i.e., computation time/communication time) of each D&C tree at Solve stage. The source code instrumentation of the experiment is specified in accordance with the event expression of Solve in Figure 5.1. Hercule classifies computation and communication cost and finds that the D&C tree originally rooted at process 2 performs worst. Then Hercule investigates the communication cost associated with D&C tree 2 in detail. Of course, any D&C tree can be identified for additional study. Hercule computes model-specific performance metrics and distinguishes load-migration communications and find-idle-processor communications, each is presented as the percentage the category contributes to the overall communication time. It then tries to explain the high cost associated with these two types of communications respectively and identifies possible factors giving rise to the communications. The inference process and diagnosis results, as shown above, are presented in a manner close to programmer's reasoning and understanding.

5.2 Diagnosing Master-worker Programs

5.2.1 Knowledge Engineering for M-W Model

A widely used parallel computation pattern is the classic Master-Worker (M-W) model. Master-Worker models a computation that is decomposed into a number of independent tasks of variable length. A *master* is responsible for assigning the tasks to a group of *workers*. Communications are required between the master and workers before and after processing each task. The workers are independent to one another. A M-W illustration with a master and two workers is displayed in Figure 5.3

The master usually employs certain task scheduling algorithms to achieve load balance and minimize makespan. M-W performance factors we identified, through performance observation of M-W codes and knowledge obtained from expert performance analysts, are:

- *Inherent sequential code fragments in the master.* In Figure 5.3, both the master and the workers need to do initialization and finalization. But the master usually spends more time in these two phases than the workers for extra management work of reading in and preprocessing the data, or post-processing worker results. The differences in initialization and finalization workload incur inherently sequential penalty at the worker side since they are dealt with exclusively by the master.

- *Number and complexity of tasks assigned to the workers.* This characterizes the total amount of task workload to be shared by all workers.

- *Task setup costs in the master and the task scheduling method.* This reflects how fast the master is able to process worker requests.

- *Number of worker processors.* It can happen that increasing the number of worker processors does not reduce the total elapsed time of execution [34]. This is because there is often a saturation point beyond which more workers cannot be effectively served by the master. This results in the master being seen as a performance bottleneck.

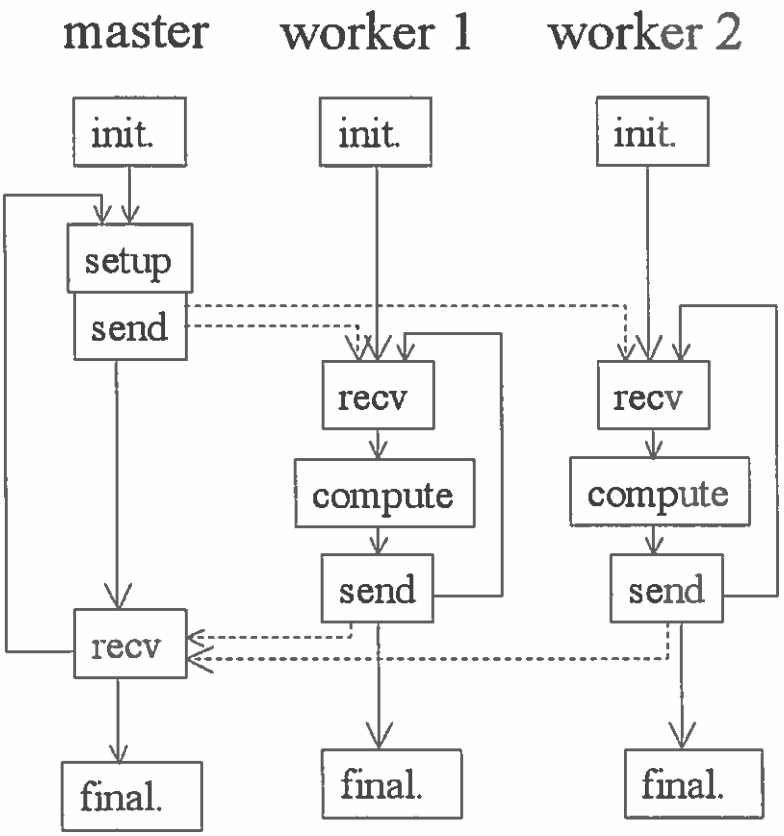


FIGURE 5.3: An illustration of Master-Worker pattern with a master and two workers.

- *Task scheduling strategy.* Scheduling independent tasks with m identical machines ($m \geq 2$) and a single server has been shown to be a NP-hard problem [35]. Often certain scheduling heuristics are used in a M-W program. If during an execution tasks are assigned in such a way that new-task requests from some workers arrive at the master at approximately the same time, then some workers have to wait while the peer requests are being processed. This is one type of inefficient scenario. Another inefficient scenario happens when a long task is assigned to the last job requester, which makes all other finished workers wait for the worker which gets this task assignment to complete computing. Avoiding the situations of master bottleneck and time-imbalance is key to achieving better worker efficiency.

In a M-W program, an independent task assigned to a worker process has a well-defined life cycle: first the worker sends a task request to the master, the master receives the request

and sets up a task, it then transfers the data and task specification to the requesting worker, and the worker processes the task until finished. At that time, that worker returns the result to the master and the cycle continues until the worker is instructed to terminate. We specify the program behaviors and performance properties associated with a task life cycle by an abstract event type **TaskLifeCycle**, as shown in Figure 5.4. Note that information in the shaded area in the figure is not available until an implementation of the model is provided. This is because the binding of appropriate values to performance properties and performance properties evaluation rules are dependent on model implementation. For instance, when using non-blocking communication routine "MPI_Irecv" instead of blocking version "MPI_Recv", the computing rule of communication time will reflect corresponding "MPI_Wait" routine as well, rather than looking at the message-receiving routine only.

FIGURE 5.4: An abstract event description of Master-Worker model. The shaded areas are instantiated by a model implementation.

Given the program behavior, we can formulate M-W performance models. Following phase localization rule, a worker's total elapsed time t_{worker} consists of t_{init} (initialization cost), t_{comp} (the amount of time spent computing tasks), t_{comm} (the amount of time spent communicating with the master), t_{wait} (the amount of time spent waiting for task assignment or synchronizing with other workers before finalization, excluding communication overhead), and t_{final} (finalization cost):

$$(\mathbf{M2}) \Rightarrow t_{worker} = t_{init} + t_{comp} + t_{comm} + t_{wait} + t_{final} \tag{5.1}$$

Whenever we refer to communication time, we mean effective message passing time that excludes time loss due to communication inefficiencies such as late sender or late receiver in MPI applications. Rather, waiting time accounts for the communication inefficiencies with the purpose of making explicit performance losses attributed to mistimed processor concurrency.

The master's total elapsed time is:

$$(\mathbf{M1}, \mathbf{M2}) \Rightarrow t_{master} = t_{init} + t_{setup} + t_{comm} + t_{idle} + t_{final} \tag{5.2}$$

Performance coupling of a worker with the master and the rest of peer workers manifests four performance overheads – t_{seq} (the master initialization and finalization costs translated to idle overhead in the worker), $t_{w-setup}$ (master task setup time), t_{w-bn} (blocking time in master bottlenecks), and $t_{w-final}$ (the cost of synchronization with other workers for finalization).

$$(\mathbf{M2}, \mathbf{M3}) \Rightarrow t_{wait} = t_{seq} + t_{w-setup} + t_{w-bn} + t_{w-final} \tag{5.3}$$

The above performance models enable us to define performance metrics specifically tailored to M-W programs. We start with evaluating individual *worker efficiencies* to detect a top-level symptom because efficiency is a reflection of total worker scalability.

$$\textbf{worker efficiency} := \frac{t_{comp}^{worker}}{t_{worker}} \tag{5.4}$$

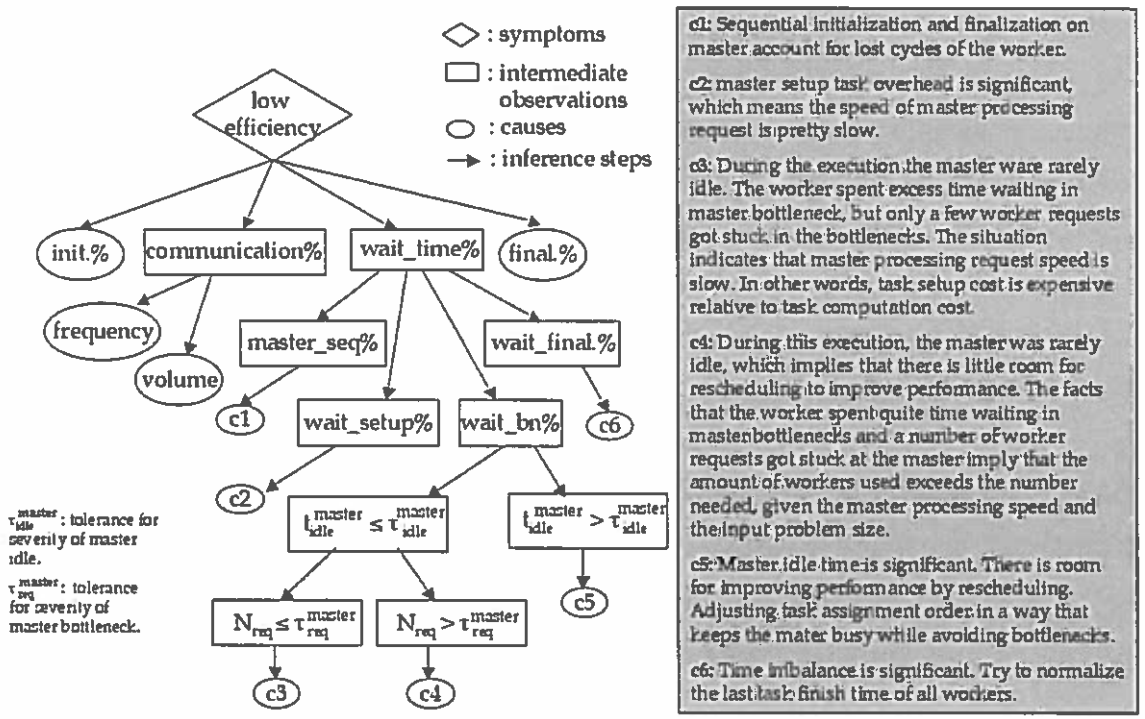


FIGURE 5.5: Inference Tree for Performance Diagnosis of M-W programs.

Refining each item in model (5.3), we obtain metrics of worker wait time:

$$
\begin{aligned}
t_{seq} &:= \max\{t^{master}_{init} - t^{worker}_{init}, 0\} + \max\{t^{master}_{final} - t^{worker}_{final}, 0\} \\
t_{w-setup} &:= \sum_{i=1}^{M} t^{i}_{setup}, \qquad t_{w-bn} := \sum_{i=1}^{M} t^{i}_{w-bn} = \sum_{i=1}^{M} (t^{i}_{wait} - t^{i}_{setup}) \\
t_{w-final} &:= \max_{all\ workers}\{T_{fin}\} - T_{fin}
\end{aligned}
$$

where M is the number of tasks the worker processes altogether, t^{i}_{setup} the amount of time for setting up task i, t^{i}_{w-bn} is the waiting time due to master bottleneck when requesting the ith task, t^{i}_{wait} is the total amount of worker idle time between sending out request and receiving task i, $\max_{allworkers}\{T_{fin}\}$ is the finish timestamp of the last task computed, and T_{fin} the last task finish timestamp of the observed worker processor.

Having been prepared with performance attributes in abstract event descriptions, we are able to reduce model-specific metric computing to aggregating attribute values of related event instances. In performance debugging, it is desirable to be able to concentrate

```
(defrule assert-masterComputeSig
  (declare (salience -30))
  (masterAssignTime ?masterAssignTime)
  (threshold_MA ?threshold_MA)
  (test (>= ?masterAssignTime ?threshold_MA))
  =>
  (assert (masterComputeSig))
)

(defrule assert-masterComputeEffect
  (declare (salience -30))
  (minWEID ?id)
  (waitingTimeSig)
  (masterComputeSig)
  (masterAssignTime ?masterAssignT)
  =>
  (assert (master-assign-task-time-sig))
)
```

FIGURE 5.6: Clips implementation of c2 asserting master computation time (for setting up task assignment) is a cause.

on and evaluate a specific code section, or a problematic execution phase as bug searching proceeds. This is attainable in our approach by gathering event instances occurring in the interested spatial or temporal regions and synthesizing their performance attribute values. For instance, to compute amount of time a worker process spends waiting for task assignments throughout the execution, we can simply add up waiting time for each task, which is attribute *WorkerWaitingTimeForTheTask* of the abstract event **TaskLifeCycle** in Figure 5.4. In this way our approach allows flexible definition of performance metrics at different abstraction levels.

Now we can incorporate these performance factors and metrics in diagnosis inference rules. An inference tree is created for every symptom type. The inference tree for explaining low efficiency of a worker process, for instance, is shown in Figure 5.5. Figure 5.6 illustrates the resulting CLIPS implementation of the assertion that master task setup time is a potential problem cause.

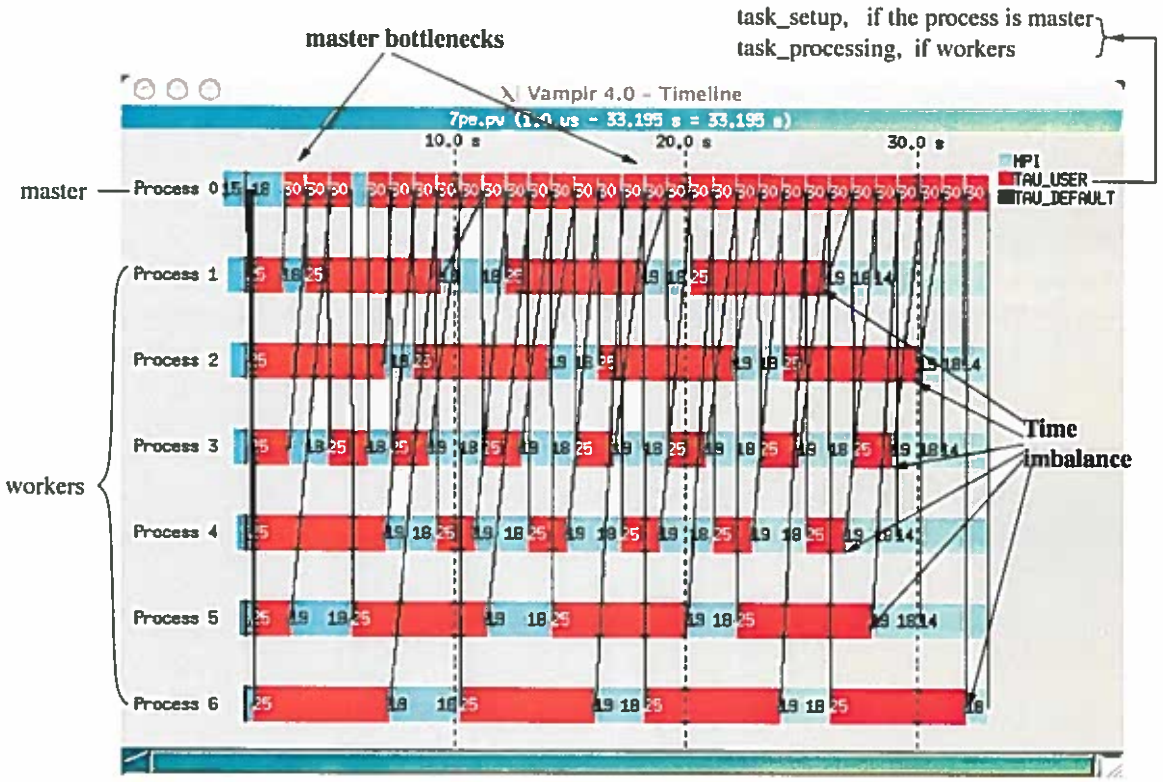


FIGURE 5.7: Vampir timeline view of an example M-W program execution.

5.2.2 Experiment with M-W Program

We tested Hercule's performance diagnosis capability for the M-W model using a synthetic M-W application. This allowed us to introduce various known performance problems (i.e., performance faults) and evaluate whether Hercule would be able to discover them. All experiments were run on an distributed memory Pentium Xeon cluster running Linux.

Using the M-W computational model components, as presented in Figure 5.3, we created a synthetic parallel program to run on our test cluster. The performance problems we introduce in the program focus on the impact of master-processing-request speed on overall performance. We implement the M-W program using MPI, and set the initialization and finalization cost of both the master and workers to a negligible value. Some number of independent tasks is chosen and their processing times are assigned during execution. The master setup task time is set to be proportional to the average task processing time. Figure 5.7 and 5.8 respectively present a Vampir [36] timeline view and ParaProf [4, 37] profile

FIGURE 5.8: Paraprof graphical display of relative time spent in each function on each node, context, thread in the M-W experiment.

display of an execution of the program with 7 processors. The event trace and profiles are generated by the TAU [4] performance measurement system with only major model components being instrumented. In Figure 5.7, red regions represent task setup at the master and task processing at the workers. Light blue regions represent MPI function calls, including **MPI_Init**, **MPI_Send**, **MPI_Recv**, and **MPI_Finalize**. Note that in both figures, blocking/waiting time of processors is implicitly included in elapsed time of blocked **MPI_Send**, **MPI_Recv** and **MPI_Finalize** operations.

Given the program and performance knowledge associated with M-W model, Hercule automatically requests three experiments during the diagnosis of this problem. The inference process and diagnosis results of these experiments are presented in Figure 5.9. The first experiment collects data for computing efficiencies of each worker. The measurement data shows that worker 3 performs worst. Then Hercule investigates the performance loss of worker 3 (of course, any worker can be identified for additional study), and issues the

TABLE 5.2: Metric values of the M-W program.

Metric name	Performance loss%
t_{w-bn}	39.2%
$t_{w-setup}$	34.3%
$t_{w-final}$	14.8%
t_{comm}	6.2%

```
dyna6-166:~/PerfDiagnosis lili$ ./model_diag MW.clp
Begin diagnosing ...
================================================================
Level 1 experiment - collect data for computing worker
efficiencies.

----------------------------------------------------------------
Worker 3 is least utilized, whose efficiency is 0.385.
================================================================
Level 2 experiment - collect data for computing
initialization, communication, finalization costs, and
wait (idle) time of worker 3.

----------------------------------------------------------------
Waiting time of worker 3 is significant.
================================================================
Level 3 experiment - collect data for computing individual
waiting time fields.

----------------------------------------------------------------
Among lost cycles of worker 3, 14.831% is spent waiting for
the last worker to finish up (time imbalance).

----------------------------------------------------------------
Master processing time for assigning task to workers is
significant relative to average task processing time, which
causes workers to wait a while for next task assignment.
Among lost cycles of worker 3, 34.301% is spent waiting
for master computing next task to assign.

----------------------------------------------------------------
Among lost cycles of worker 3, 39.227% is spent waiting
for the master to process other workers' requests in
bottlenecks. This is because master processing time for
assigning task is expensive relative to average task
processing time, which causes some workers to queue
up waiting for task assignment.
================================================================
Diagnosing finished...
```

FIGURE 5.9: Diagnosis result output from Hercule of the M-W test program.

second experiment to evaluate individual overheads in equation (5.1). Waiting time cost stands out as a result of this inference step. The third experiment then targets performance loss categories in equation (5.3). Table 5.2 presents model-specific metrics computed during the diagnosis in the form of percentage that each overhead category contributes to the overall performance loss (i.e., total elapsed execution time minus effective task processing time). It is important to note that diagnosis results can be encoded to present output in a manner close to programmer's reasoning and understanding of the M-W computation model.

5.3 Wavefront (Pipeline) Model and Relative Diagnosis of Sweep3d

Wavefront is a two-dimensional variant of a traditional pipeline pattern. Computation or data is partitioned and distributed on a two-dimensional process grid where every processor receives data from preceding processors and passes down data to successive processors in two orthogonal directions. Those processors within each wavefront, i.e., those on a diagonal, are algorithmically independent and can do computations concurrently. The data dependence of wavefront parallelism is shown in Figure 5.10. Additional concurrency can be achieved by blocking the computation, resulting in more wavefront sweeps using smaller data sets. Well-known pipeline performance problems include sensitivity to load imbalance, processor idleness when pipeline fills up and empties, and so on. It is these types of problems that we want to find. Next we will show how performance knowledge for diagnosing Wavefront programs is engineered with model-based approach.

5.3.1 Knowledge Engineering for Wavefront Model

The abstract event describing a Wavefront process node is shown in figure 5.11.

According to the behavioral descriptions of abstract event, a processor's execution time t_{proc} can be decomposed into t_{init} (initialization cost), t_{comp} (the amount of time spent computing tasks), t_{comm} (the amount of time spent communicating with the neighbor pro-

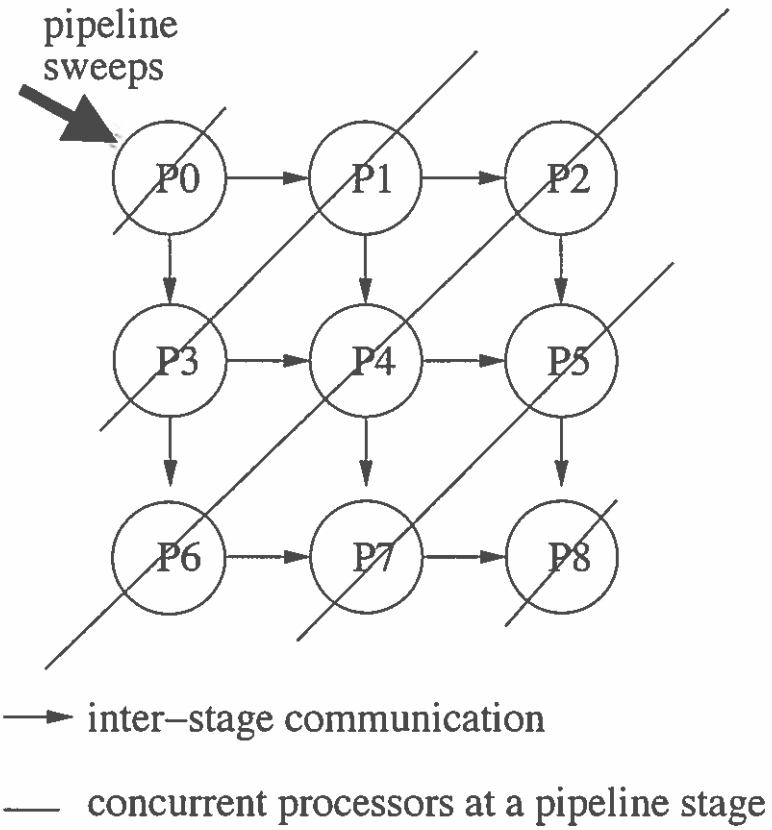


FIGURE 5.10: Wavefront parallelism on a 3x3 process grid. Each node represents a processor in this grid.

cessors in the pipeline), t_{wait} (the amount of idle time at the communication points), and t_{final} (finalization cost):

$$(\mathrm{M2}) \Rightarrow t_{proc} = t_{init} + t_{comp} + t_{comm} + t_{wait} + t_{final} \tag{5.5}$$

Performance coupling of a processor with neighbors in the pipeline manifests four performance overheads – t_{pl_fill} (the waiting time for the pipeline to fill up at the pipeline start-up), t_{pl_empty} (the waiting time for the pipeline to empty up at the end of pipeline computing), $t_{pl\_handshake}$ (the waiting time to receive data from predecessor processors in sweeps), and $t_{pl\_directionchange}$ (the performance penalty due to pipeline sweep direction change).

$$(\mathrm{M2}, \mathrm{M3}) \Rightarrow t_{wait} = t_{pl\_fill} + t_{pl\_empty} + t_{pl\_handshake} + t_{pl\_directionchange} \tag{5.6}$$

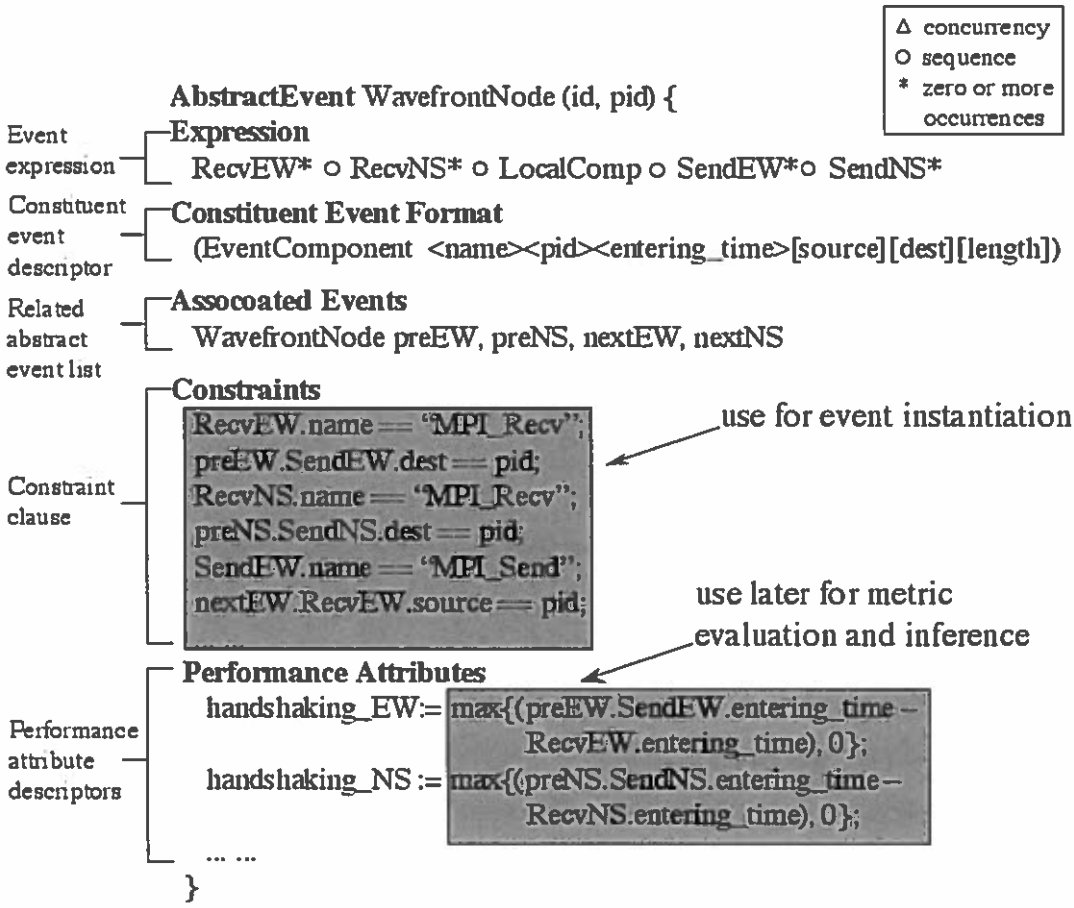


FIGURE 5.11: An abstract event type description of Wavefront model. The shaded areas are instantiated by an algorithmic implementation of the model.

Performance metrics tailored to Wavefront model can be derived from the performance models. For instance, we define t_{pl_fill} as:

$$t_{pl\_fill} = \sum_{i=1}^{M-1} t_{pre-proc}^{comp.,i}$$

where M represents the pipeline stage number where the processor is located, and $t_{pre-proc}^{comp.,i}$ the computation time of the preceding processors at the ith stage during pipeline start-up.

Now we can incorporate these performance metrics into diagnosis inference rules. The inference tree for explaining low speedup of wavefront computing, for instance, is shown in Figure 5.12.

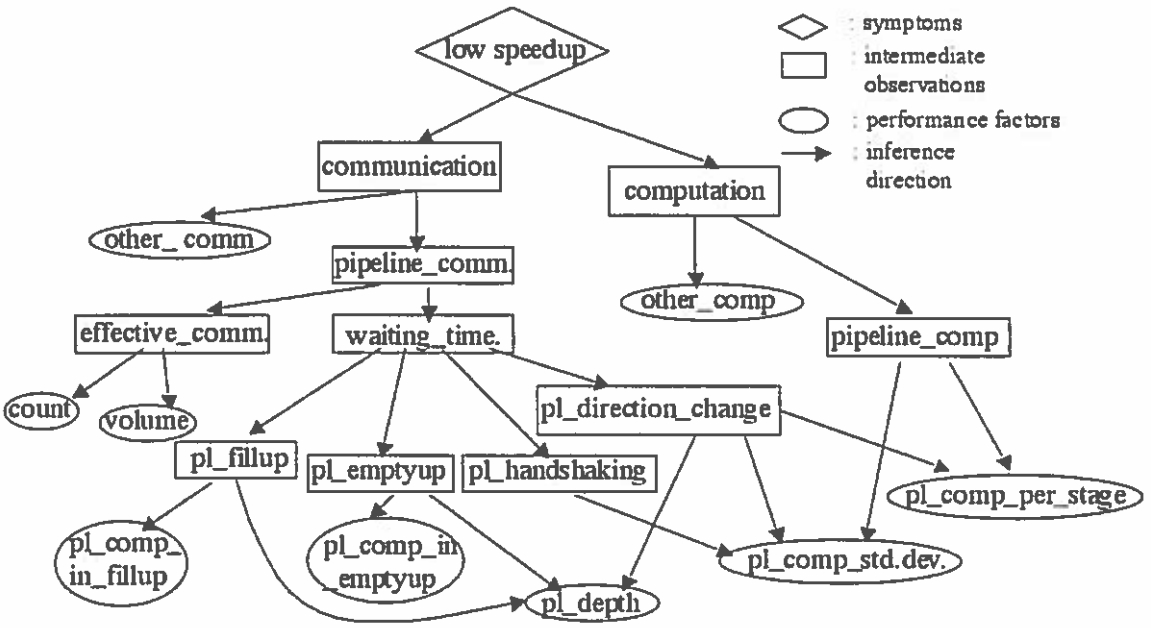


FIGURE 5.12: An inference tree of Wavefront model that diagnoses low-speedup

5.3.2 Relative Performance Diagnosis of Sweep3D

In this section, we will demonstrate Hercule's effectiveness in *relative performance diagnosis* of the ASCI Sweep3D benchmark. Sweep3D [11] is a solver for the 3-D, time-independent, neutron particle transport equation on an orthogonal mesh. It uses Wave-front computational model. Its parallelism comes from multiple wavefronts in multiple dimensions, which are partitioned and pipelined on a distributed memory system. The three-dimensional space is partitioned on a two-dimensional processor grid, where each processor is assigned one columnar domain. Sweep3D exchanges messages between adjacent processors in the grid as wavefront propagates diagonally across the 3-D space in eight directions.

Sweep3D is a well-researched parallel benchmark. Although parallelism overheads in Sweep3D have been minimized, for instance, by evenly distributing data across a process grid, leaving little room for performance tuning, Hercule can tell exactly how running time is spent in terms of model semantics, helping understand inherent performance losses of the model under an optimistic condition. Our performance study with Sweep3D focuses on overall scalability, looking at how well the application scales as the number of processors

is increased (strong scaling) and as total problem size increases with the process count increase (weak scaling).

Diagnosis of a single execution is incomplete as a comprehensive diagnosis process. Understanding of performance problems routinely involves *comparative* and *relative* interpretation. In the experiments with Sweep3D, we improve the Hercule methodology to support what we will term *relative performance diagnosis* (in the spirit of relative debugging [38]). The multi-experiment relative performance analysis is also addressed in [39] [40] etc.

Understanding of performance problems routinely involves *comparative* and *relative* interpretation. Performance analysts often need to answer such questions in scalability analysis of a parallel application: what are most pronounced performance differences between two program executions with difference problem scales, which program design factors contribute to the differences, and what are magnitudes of their contributions?

Hercule's single execution diagnosis can be extended to support what we term *relative performance diagnosis* that is intended to answer the questions. To interpret what was happening at the performance anomalies with certain problem scale, we pick a performance reference run, in the family of scalability executions, which has comparatively normal performance and evaluate problematic runs against it. Relative performance diagnosis follows the same inference processes as presented in model-specific inference trees except for performance evaluation at branch nodes. Recall that cause inference in the inference trees is driven by performance evaluation, that is, to compare the model-specific metric with an expected value (from performance modeling) to decide on an intermediate observation. In relative performance diagnosis, we calculate the expected value based on model-specific metrics of the reference run to evaluate problem behaviors. Examples of relative diagnosis of anomalous Wavefront application executions will be presented in the next section.

Hercule extensions for supporting relative performance diagnosis manifest in the interfacing of the metric evaluator and the inference engine. To assert the performance observation associated with a branch node in the inference tree, the metric evaluator takes in event instances of two runs to be compared and feeds the calculated model-specific metrics into the inference engine. The inference engine sets a performance expectation according to the reference run metric and evaluates the problematic run against it.

We ran Sweep3D tests on MCR, a linux cluster located at Lawrence Livermore national Laboratory. MCR has 1,152 nodes, each with two 2.4-GHz Pentium 4 Xeon processors and 4 GB of memory and has peak performance rating of 11.06 Tflop/s. The system interconnection is a customized 1024-port single rail QsNet network, which provides a 400 MBytes/s bi-directional bandwidth.

Case I: Diagnose strong scaling performance problems

Figure 5.13 shows the strong scaling behavior of Sweep3D with problem size 150^3, and angle blocking factor, *mmi*, equal to 3, k-blocking factor, *mk*, equal to 10. The application scales well in general, but at process count 32 the speedup drops and bounces up when process count increases to 36. We applied Hercule to contrast the performance of run1 (with 32 processors) against run2 (with 36 processors) and diagnose performance anomaly causes. Hercule uses relative speedup (compared to two-processor run) to evaluate performance since there is no inter-processor communication in a sequential execution. The results that follow were generated in a completely automated manner.

Hercule first calculates speedup of run1 (with 32 processors), run2 (with 36 processors) relative to run3 (with 2 processors), and expected speedup of run1 based on run2 performance. It reaches a performance symptom of run1 that will be further explained.

Hercule diagnosis step 1: find performance symptom

```
dyna6-166:~/PerfDiagnosis lili$ ./model_diag WF_speedup.clp
32pe.dup 36pe.dup 2pe.dup

Begin diagnosing ...
============================================================
Speedup of run1 and run2 relative to run3
                run1        run2        expected run1
speedup        12.80        15.84       14.08
------------------------------------------------------------
run1 is slower than the expected value 14.08
------------------------------------------------------------
Next we look at the symptom low speedup.
============================================================
```

Hercule then breaks runtime down into computation and communication, narrowing performance bug search.

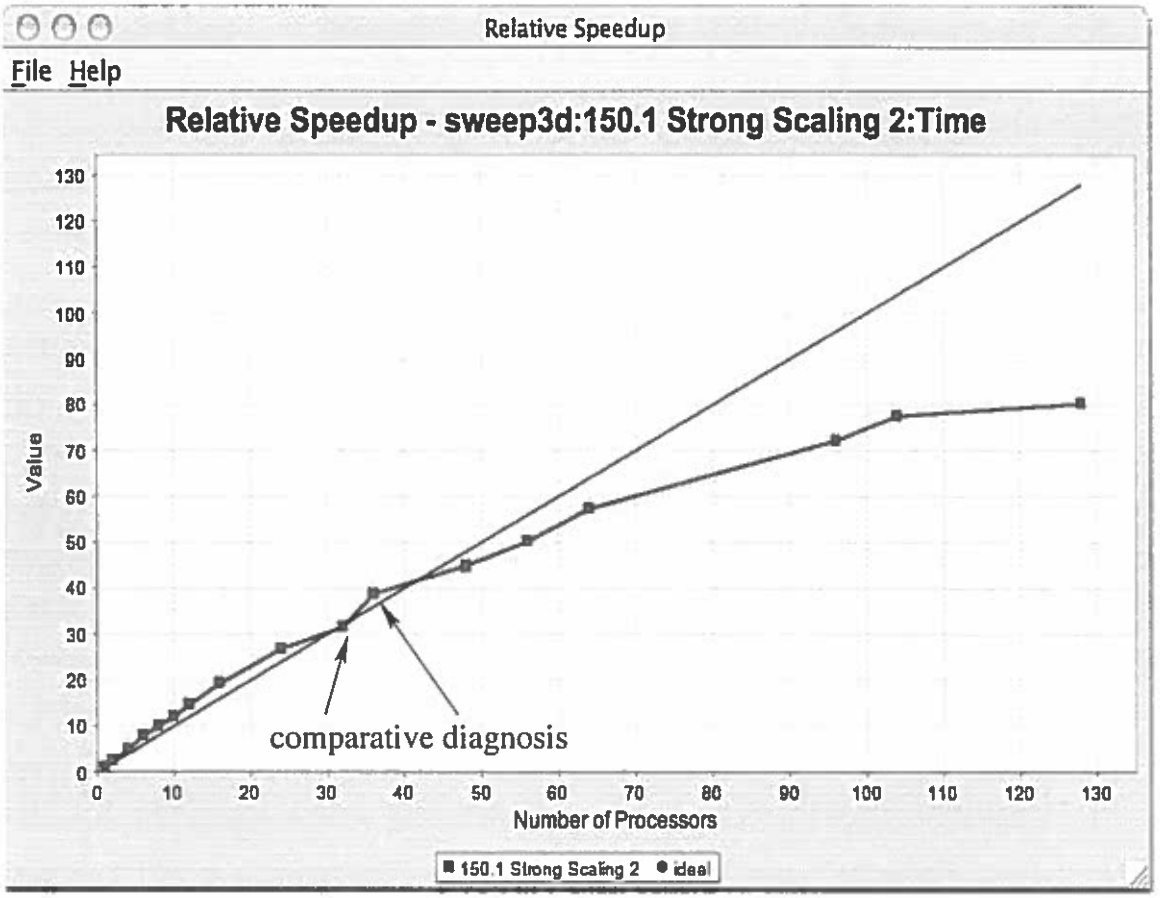


FIGURE 5.13: Sweep3D strong scaling with problem size 150x150x150 (mmi=3, mk=10)

Hercule diagnosis step 2: locate poorly performed functional groups

```
============================================================
Level 1 experiment -- collect performance data with respect
to comp. and comm..

------------------------------------------------------------
Relative speedup of functional groups in run1 and run2
                      run1          run2     expected run 1
computation:        16.035        19.906          17.694
communication:       1.115         1.172           1.042

------------------------------------------------------------
computation in run1 is longer than the expected.

------------------------------------------------------------
Next look at performance with respect to pipeline
components.
```

As computation time per process stands out, Hercule further distinguishes pipeline-related computation and others.

Hercule diagnosis step 3: refine locating poorly performed functional groups

```
============================================================
Level 2 experiment -- collect performance data with respect
to pipeline components.

------------------------------------------------------------
Relative speedup of pipeline components in run1 and run2
                            run1      run2   expected run 1
computation in pipeline:  16.598    20.702    18.402
other computation:        10.452    12.405    11.03

------------------------------------------------------------
computation in pipeline in run1 is slower than the expected
most.

------------------------------------------------------------
Next look at computation in pipeline.
============================================================
```

Pipeline computation per process in run1 is more expensive than the expected. Hercule then looks at how well the load is distributed on processes.

Hercule diagnosis step 4: form performance hypothesis

```
============================================================
                                      run1    run2  difference
computation in pipeline SDV (us): 236859  97548  139311
(w.r.t. processes)

------------------------------------------------------------

Standard deviation of pipeline computation in run1 is
significantly larger than run2, which implies a load
imbalance across processes.

------------------------------------------------------------
Next testify the hypothesis load imbalance.
============================================================
```

Hercule forms a load imbalance performance hypothesis based on the standard deviation of pipeline computations on all processes. It tests the hypothesis by looking at model-related overheads to which load imbalance possibly contributes most. It calculates and distinguishes performance impact of load imbalance on the overhead categories, and exemplifies occurrence of load imbalance with process behaviors in some specific computation

step (iteration) and pipeline sweep. This way of explanation provides the users with both the nature of performance causes and evaluations of performance impact of the causes.

Hercule diagnosis step 5: test performance hypothesis

```
==========================================================
The impact of process load imbalance on performance
manifests in pipeline-handshaking and
sweep-direction-change overhead.

Passing along data among successive pipeline stages
(handshaking) takes 14.9% of pipeline communication time.
Pipeline handshake delay is unevenly distributed across
processes. std dev = 486463.75. process 31 involves the
longest pipeline handshake cost.

----------------------------------------------------------
Level 3 experiment for diagnosing handshaking related
problems -- collect performance event trace with respect
to process 31

----------------------------------------------------------
Pipeline HS delay is evenly distributed across iterations
in the process 31. Next we look at performance
characteristics of iteration 3 which involves the longest
pipeline HS.

Pipeline HS delay is evenly distributed across sweep in
iteration 3 process 31, Next we look at sweep 6 which
involves the longest pipeline HS.

In iteration 3 sweep 6, computation are unevenly
distributed across pipeline stages. For example,
in stage 4 process 4 spends 1964(us) doing
computation, while in stage 10 process 31
spends 1590(us) computing.

In general, process 31 is assigned 23.6% less work
load than process 4. Such discrepancy causes process
31 idle for 29.5% of pipeline communication time..

----------------------------------------------------------
When pipeline sweep direction change, processes may be idle
waiting for successive pipeline stages in previous sweep to
finish, and for pipeline to fill up in a new sweep. The
sweep direction changes comprise 34.6% of pipeline commu.
```

```
time. The delay is unevenly distributed across processes.
process 31 involves the longest pipeline direction change.

-----------------------------------------------------------
Level 3 experiment for diagnosing sweep-direction-change
related problems -- collect performance event  trace with
respect to process 31
-----------------------------------------------------------
Pipeline direction change delay of process 31 is unevenly
distributed across iterations. Next we look at performance
of the iteration 10 which involves the longest direction
change delay.

In this wavefront execution, pipeline sweep direction
change delay is significant in process 31, especially in
iteration 10. Between sweep 3 and 4, process 31 has been
idle for 117980(us). Among the idle time, 85.5% is spent
waiting for successive pipeline stages in sweep 3 to finish
up, and 14.7% waiting for pipeline filling up in sweep 4.
We compare performance behaviors in process 31 and
next sweep head (process 24) to explain where is the idle
time from.

In sweep 3 process 31 is in pipeline stage 3, next sweep
head, process 24, is in stage 10. Due to the pipeline
working mechanism, process 31 has to wait process 24
to finish computation before next sweep begins.
Computation load difference between the two processes,
by 12.8%, contributes 39.9% to the direction change delay.
===========================================================
Diagnosing finished...
```

Case II: Diagnose weak scaling performance problems

The second experiment with Hercule demonstrates its capability of identifying and explaining parallelism overhead increases as both problem size and process count are increased in weak scaling study. Figure 5.14 shows the weak scaling behavior of Sweep3D with fixed problem size 20x20x320. We can see that runtime increases as more processors are used even though each process's computation load is kept the same. Hercule will compare 4-processor and 48-processor run and report and explain the performance difference. Again, the results that follow are generated in a completely automated fashion.

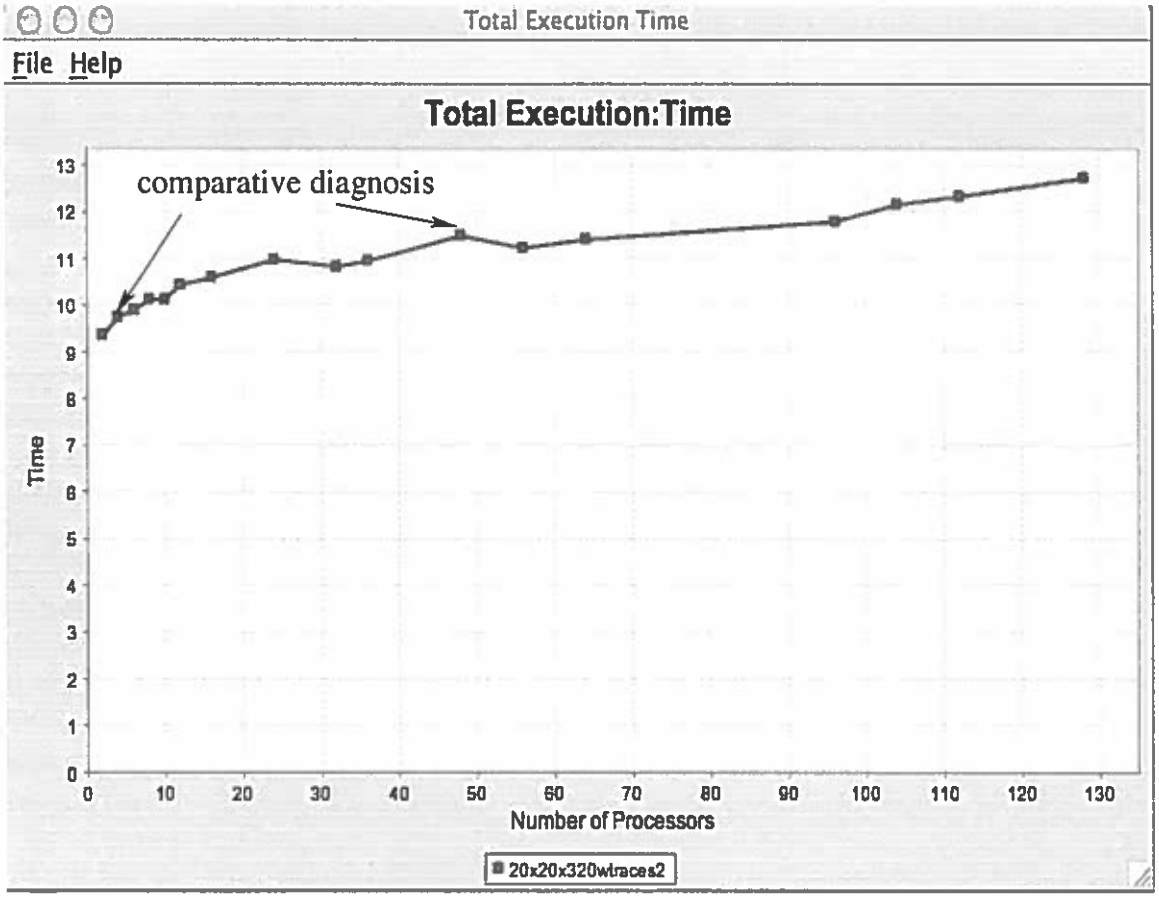


FIGURE 5.14: Sweep3D weak scaling with problem size 20x20x320 (mmi=3, mk=10)

Hercule first calculates significance of performance difference and reaches a performance symptom, higher parallelism overhead.

Hercule diagnosis step 1: find performance symptom

```
dyna6-166:~/PerfDiagnosis lili$ ./model_diag WF_overhead.clp
weak.48pe.dup weak.4pe.dup

Begin diagnosing ...
================================================================
Runtime of run1 and run2 (in seconds)
                        run1         run2        difference%
runtime                11.489       9.815         17.055%
```

```
run1 is 17.055% slower than the run2.

Next we look at the symptom parallelism overhead.
```

Hercule then breaks runtime down into computation and communication, locating the functional group with most pronounced performance difference.

Hercule diagnosis step 2: locate poorly performed functional groups

```
Level 1 experiment - collect performance data with respect
to comp. and comm..

Runtime of functional groups in run1 and run2 (in seconds)
                      run1          run2        difference%
computation:          8.886         8.891       -5.624e-4
communication:        2.603         0.924       181.71%

commu. cost in run1 is significantly higher than run2.

Next look at communication performance with respect to
pipeline components.
```

Hercule further distinguishes pipeline-related communication and others.

Hercule diagnosis step 3: refine locating poorly performed functional groups

```
Level 2 experiment -- collect performance data with respect
to pipeline components.

Runtime of pipeline in run1 and run2 (in seconds)
                                  run1     run2   difference%
computation in pipeline:         8.014    8.013    1.25e-4
other computation:               0.872    0.878    -6.83e-3

communication in pipeline:       2.275     0.803     183.31%
effective commu. in pipeline:    0.943     0.571     65.15%
waiting time in pipeline:        1.332     0.231     476.62%
other communication:             0.328     0.121     171.07%

comm. count (count/per process): 12288     12288        0
```

```
comm. volume (byte/per process): 58982400 58982400    0

Pipeline waiting time in run1 is 476.62% higher than run2.

Next look at pipeline overheads.
```

Since waiting time in pipeline is significant, Hercule refines model-specific overhead categories and computes corresponding metrics.

Hercule diagnosis step 4: locate poorly performed pipeline components

```
Level 3 experiment -- collect performance data with respect
to pipeline waiting time

Runtime of pipeline components in run1 and run2 (in seconds)
                                run1    run2   difference%
 waiting time in pipeline       1.332   0.231   476.62%
 pipeline fill-up:              0.161   0.014   1050%
 pipeline empty-up:             0.244   0.017   1335.29%
 pipeline handshaking:          0.337   0.075   349.33%
 pipeline direction change:     0.584   0.125   367.2%
```

There are increases in most of overhead categories. We present below diagnosis results explaining two most pronounced categories, pipeline fill-up and empty-up.

Hercule diagnosis step 5: diagnose two most pronounced pipeline overheads

```
Diagnosing pipeline fill-up ... ...

In run1, pipeline fill-up delay is evenly distributed over
iterations. We look at performance characteristics of the
iteration 0, which involves the longest pipeline fill-up.

In iteration 0, the depth of pipeline is 13. The pipeline
tail, process 0 is being idle when the pipeline is filling
up by processes in preceding stages. The pipeline fill-up
```

delay comprises 335103us (20.8%) of process 0's total waiting time. The computations at preceding pipeline stages together account for the long waiting time at the process. Reducing computation load at preceding stages or pipeline depth will decrease filling up time.

In run2, pipeline fill-up delay is evenly distributed across iterations. We look at performance characteristics of the iteration 1, which involves the longest pipeline fill-up.

In iteration 1, the depth of pipeline is 3. The pipeline tail, process 0 is idle while the pipeline is filling up by processes in preceding stages. The pipeline fill-up delay comprises 28707us (25.5%) of process 0's total waiting time.

Diagnosing pipeline empty-up ... ...

In run1, pipeline empty-up delay is evenly distributed across iterations. Next we look at performance characteristics of the iteration 4, which involves the longest pipeline empty-up.

In iteration 4, the depth of pipeline is 13. The pipeline head, process 0 is being idle when the pipeline is emptying up by processes in successive stages. The pipeline empty-up delay comprises 573,162us (35.5%) of process 0's total waiting time. The computations at successive pipeline stages together account for the long waiting time at the process. Reducing workload at successive pipeline stages or pipeline depth will decrease empty-up time.

In run2, pipeline empty-up delay is evenly distributed across iterations. We look at performance characteristics of the iteration 1, which involves the longest pipeline empty-up.

In iteration 1, the depth of pipeline is 3. The pipeline head, process 0 is being idle when the pipeline is emptying up by processes in successive stages. The

```
pipeline empty-up delay comprises 34858us (31.0%)
of process 0's total waiting time.
============================================================
```

As shown in the results, the increase of pipeline depth in run1 (48-processor run) is clearly the main cause of its overhead increase. Hercule illustrates and interprets performance impact of the pipeline depth with the behaviors of the process of the longest pipeline fill-up and empty-up. The pipeline depth also has a performance effect on sweep direction change. For the purpose of brevity we skip the interpretation of other overhead categories, event though Hercule is able to explain them equally well.

5.4 Chapter Summary

In this chapter, We presented Hercule experiments with three parallel applications that represent Divide-and-Conquer, Master-Worker, and Wavefront model respectively. Our experience diagnosing the programs shows that model-based performance knowledge can provide effective guidance for locating and explaining performance bugs at a high level of program abstraction. Particularly in Wavefront analysis, we extend Hercule to allow for relative execution diagnostic analysis that compare multiple experiment performance and explain the differences with model semantics. Relative diagnoses of Sweep3D (implemented with Wavefront model) performance anomalies in strong and weak scaling cases are given. This broadens the application of model-based performance diagnosis approach to multi-experiment investigation.

CHAPTER VI

COMPOSITIONAL MODEL DIAGNOSIS

In the previous chapters, we focused on generating and encoding performance knowledge from singleton models, and we developed the *Hercule* performance diagnosis system to validate how performance knowledge derived from parallel models provides a sound basis for automating performance diagnosis processes and explaining performance loss from high-level computation semantics. This has been shown for several parallel models to date (e.g., master-worker, divide-and-conquer, and pipeline). However, we also realized that singleton model program analysis is incomplete as a comprehensive diagnosis process. Often parallel programmers in scientific computing combine two or more computational models to realize the intended parallelism or address specific performance issues. *Compositional* models capture how singleton models are composed together and interact in a parallel program. The model composition may change performance identify of individual models. This raises new challenges as to how to allow the amount and location of the occurred performance loss due to model interaction to be detected and interpreted within an integrated environment. This chapter reports our experience with investigating performance problems of compositional models. We will present a framework for discovering and interpreting performance bugs using both the semantics of individual models and their composition pattern. We extend our Hercule performance diagnosis framework to support performance engineering of compositional models and test Hercule on the scientific application FLASH [41] and the ScaLAPACK algorithm PDLAHQR [42], each representing different types of model integration. These experience demonstrate that our approach can effectively support automatic diagnosis of compositional model performance.

In Section §6.1, we classify different model composition patterns and highlight their impact on performance. We then present in Section §6.2 the performance knowledge engineering approach adaptive to the model compositions. Hercule is extended to support the compositional diagnostic analysis, which is presented in Section §6.3. We show in Section §6.4 the Hercule experiments with two real-world applications that represent different types of model integration. Section §6.5 concludes with observations.

6.1 Computational Model Composition

In the previous chapters, we investigated how to generate and encode performance knowledge from singleton models, and developed the *Hercule* system to test how effective the derived performance knowledge support for performance diagnosis processes. While the results using Hercule were generally successful, real-world applications are more complex, often based on the composition or synthesis of two or more elementary computational models. To conduct performance diagnosis of a compositional parallel program we must extend the knowledge engineering and problem inferencing to capture the interplay of one model with another. Since we have well researched single model performance, we intend to devise compositional model diagnosis approach as an extension to it.

In this section, we will look at different model composition patterns, especially their impact on performance. Parallel models can be combined in quite a few styles. Model compositions can be roughly divided into two categories from the standpoint of performance effects of the interaction. The first category is simply an addition of the involving models without cross-interleaving model components. The model composition, therefore, does not incur new performance overhead type. The second category may shuffle the model components so that performance effects of the original models change or new effects arise. Performance analysis of the second type is more complicated, and will be the focus in the following discussion.

To help understand model interaction, we follow the description format of model behavior we used in the singleton model diagnosis, abstract events. Consider a parallel com-

putational model as a set of indivisible computational components, $\{C_1, C_2, ..., C_k\}$, and a function, $F(C_1, C_2, ..., C_k)$, that specifies the relative control order (e.g., sequential, choice, concurrent, iteration, and so on) of component occurrence. We can then regard the composition of two models, $F(...)$ and $G(...)$, as an integration of the components in some manner. Model interleaving is constrained to the component level, that is, model components can be shuffled when combining two models.

Several compositional forms are possible. For instance, one model could simply nest one model hierarchically within another (model nesting), or the component sets of two models could be restructured in a more complex way by a higher-order function (model restructuring). Our objective is to understand the compositional properties of model integration in order to engineer the performance knowledge needed for performance diagnosis. Our approach will describe how performance effects of individual models change as the components merge and how new performance effects arise from the composite interactions.

6.1.1 Model Nesting

This type of model composition refines a high-level *root* (outer) model's components with lower-level *child* (inner) models so that the workload of the refined components is computed in the parallelism specified by the child models. Model nesting forms a hierarchy of models where the root model dictates the parallelism of the program at the highest level of abstraction, and the lower-level child models address parallelization and implementation of the finer-level details. State more formally, two models $F(C_1, C_2, ..., C_k)$ and $G(D_1, D_2, ..., D_l)$ (D_i are components of G) may compose into a new nested model as follows:

$$\begin{aligned} &F(C_1, C_2, ..., C_k) + G(D_1, D_2, ..., D_l) \rightarrow \\ &\quad F(C_1\{G(D_1, D_2, ..., D_l)\}, \\ &\qquad C_2\{G(D_1, D_2, ..., D_l)\}, \\ &\qquad \cdots, \\ &\qquad C_k\{G(D_1, D_2, ..., D_l)\}) \end{aligned} \tag{6.1}$$

where $C_i\{G(D_1, D_2, ..., D_l)\}$ means the component C_i implements the G model. Note, not necessarily every component in F is refined with G, and there may be additional child models used.

Parallel applications based on nested computational models are common. Iterative, multi-phase applications are frequently structured as nested models with an outer code controlling multiple phases each based on a possibly different parallel pattern. Example applications are FLASH [41], which nests parallel recursive tree into adaptive mesh refinement model, and graphical animation in [43], which implements expensive pipeline stages with master-worker model. Our concern is how to understand the performance of nested models. Due to the hierarchical structure of model nesting, analyzing this type of application usually starts with the root model. When a problematic component is found in the model (e.g., an expensive phase), we switch from the root to the component's model to refine performance problem search. The search continues until the finest level of model is reached. Performance overhead categories of the nested model is the union of the overheads associated with the participant models. Thus, they should be organized in a hierarchy conforming to the model nesting to support the top-down performance bug search.

6.1.2 Model Restructuring

The *restructuring* type of model composition integrates components of two or more models according to some new function while maintaining the same relative control order of each model's components. Formally two models $F(C_1, C_2, ..., C_k)$ and $G(D_1, D_2, ..., D_l)$ may compose into a new restructured model as follows:

$$\begin{aligned} F(C_1, C_2, ..., C_k) + G(D_1, D_2, ..., D_l) \rightarrow \\ H((\{C_1^F, ..., C_k^F\} \mid \{D_1^G, ..., D_l^G\})^+) \end{aligned} \tag{6.2}$$

where $\{C_1^F, ..., C_k^F\} \mid \{D_1^G, ..., D_l^G\}$ selects a component C_i^F or D_j^G such that the relative control order of F components and G components are maintained.

The general idea is that the components of the contributing models are being mixed to form a new set, to which a new model function H is applied. H could be F, G, or a new operation, like iteration, nesting, farming, and so on. A simple example might be the

restructuring of two pipeline models into a single pipeline model with the components at each pipeline stage merged. More complex examples are the nonsymmetric QR algorithm (PDLAHQR) [42] in ScaLAPACK, which combines pipeline and geometric decomposition models, and the MUMPS sparse direct solver [44], where parallel tree, master-worker, and geometric decomposition models are mixed together.

For our purposes, the key difference between model nesting and model restructuring has to do with the notion of *working context*. In model restructuring, the working context of a component from a contributing model will be different from its context in the singleton form of that model. The performance overheads associated with the original models (see [45, 46]), will change corresponding to context-specific factors and the new model function H. In contrast, when computational models are nested, the model semantics at each level of hierarchy will be preserved, and the working context for components of a nested model will be model-local. From a performance diagnosis perspective, the performance overheads and problem causes can thus be isolated to the models used at different levels of hierarchy.

As we have already gained performance knowledge of individual models, diagnosing a parallel program coded with restructured models basically requires we learn how performance effects of individual models change as the components interleave. Specifically, we need to identify the boundary of two models effect, that is to answer which model(s) caused a performance degradation, and how. New performance characteristics introduced by model restructuring include *delegated delay* and *composite delay*. When components of a model are separated by another model, the performance delay associated with the model may take place in the second model's code region. We call the performance delay, which is caused by a model while manifesting the effect in another model's effective context, *delegated delay*. From Equation 6.2, a delay originally caused by C_i^F and manifested in a later component C_j^F, may now appear in another component, say D_k^G occurring before C_j^F. This delay is then an example of delegated delay and should be attributed to its original model.

A *composite delay*, as the name implies, is jointly caused by two or more participant models. In this case, a performance delay associated with a model, while remaining its occurrence location inside the original model, reflects not only the original model's effect but the interleaving models. From Equation 6.2, a performance delay happening inside C_i^F, may reflect the cumulative effects of preceding F and G components, as its working

context switches from solely F (in the singleton model) to mixed F and G. To assess composite delay quantitatively, we need to change original evaluation rules in response to the model interaction. In next section, we will discuss how to incorporate the compositional performance effects into the automatic diagnosis framework.

6.2 Performance Knowledge Engineering Adaptive to Model Composition

At the core of our automatic performance diagnosis approach is engineering performance knowledge from computational models, which proceeds in four stages, from behavioral modeling → performance modeling → model-specific metric definition → inference modeling. The model-specific knowledge is stored into a base which, if interfacing to an appropriate inference engine, will support automatic performance diagnosis. A program using a customized parallel model may introduce new diagnostic requirements as to problem discovery and inferencing. Our four-step knowledge engineering approach is applicable to the program-specific knowledge as well. The user can follow the principles to extract adapted knowledge step by step and then join them with the inherited model knowledge to analyze their own programs. The model composition is another type of model variation. In this case, instead of one single model, two or more are combined together in a program. In this section, we extend our knowledge engineering approach to address performance issues arising from the model interaction, so that the user can use the approach as the guideline to "compose" performance knowledge about a compositional model from already available knowledge of individual participant models.

6.2.1 Behavioral Modeling

Behavioral modeling captures program execution semantics as behavioral models represented by a set of abstract events at varying detail levels, depending on the complexity of the model and diagnosis needs. The purpose of the abstract events in the diagnosis system is to give contextual informaton for performance modeling, metric definition, and diagnostic inferencing. An abstract event description essentially includes an *expression*

that takes the form of $F(C_1, C_2, ..., C_k)$. The expression names the constituent component C_i (typically, an indivisible computational component or communication function) and enforces their occurrence order F using event operators. Model composition may interleave constituent components of abstract events from different models.

We describe behavioral characteristics of a compositional model (or called composite events) by integrating already available abstract events of the participant models (or called basic events) in a manner that conforms to their composition style, nesting or restructuring. We use the order operators *sequential* (o), *choice* (|), *concurrent* (Δ), *repetition* (+ or *), and *occur zero or one time* ([]) to specify occurrence order of the basic events. As shown in equation 6.1 model nesting requires that a component in a root model event be replaced by a whole basic event from the child model.

Model restructuring brings up a more complicated scenario where two basic events from different models interleave components as shown in equation 6.2. In this case, we can first look at the compositional behavior and represent it with an abstract event expression without considering constituent components' semantics in their original models. Then we discern and sort out the constituent components into their original models, and annotate the components at the model switch points to distinguish the model interleaving pattern.

6.2.2 Performance Modeling and Metric Formulation

Performance modeling is carried out based on the structural information in the abstract events. The modeling leads to the formulation of *performance metrics* that represent the performance properties dictated by the model semantics. We can use the metrics to learn and evaluate various aspects of a model performance.

Performance metrics in a compositional model are not simply a union of the metrics in participant models. As we discussed in section 6.1, they may change as to their occurrence locations and evaluation rules. In the model restructuring, delegated delay and composite delay can be identified when performance modeling the composite abstract events. The annotated components at model switch points provide a clue to where a performance delay possibly transfers to. A performance loss that originally happens in a model now should take into account the interleaving model's cumulative effects in the evaluation.

6.2.3 Inference Modeling

Inference modeling captures and represents the performance bug search and interpretation process formally. Targeting performance interpretation at a high-level abstraction, we aim to find performance causes (i.e., an interpretation of a performance anomaly) at the level of parallelization design, that is, to attribute a performance problem to the culprit model factor. The performance inferencing is therefore the mapping of low-level performance measures to high-level performance factors. The inference process is captured in the form of an inference tree where the root is the symptom (i.e., a performance anomaly deviating from the expected) to be diagnosed, the branch nodes are intermediate observations obtained so far and needing further performance evidences to explain, and the leaf nodes are explanations of the root symptom in terms of high-level performance factors associated with the computational model used.

We generate the inference tree for a compositional model by merging the individual inference trees of the participant models. The inference tree merge for a nested model is based on its model hierarchy, where we expand the inference tree of the root model with relevant tree branches of the child models in the hierarchy. Recall that each node in an inference tree represents an intermediate observations that is obtained by evaluating a model-specific metric. If an involving component in a metric evaluation is refined by a new model, the sub-trees associated with the metric will be expanded with the new model's inference tree. The algorithms for merging inference trees of model nesting is presented in Appendix A. An example of tree-merging is shown in section 6.4.2 with FLASH code.

For constructing an inference tree of model restructuring, we merge inference trees of the participant models that share the same performance symptom, that is, root node. We pick one model tree as the host to expand with inference processes of a second model. The host tree is usually the one of highest complexity among the involving models. We add in the inference tree of the second model node by node – we look for the node's correct location in the host and build its connections with the host tree nodes according to the interaction pattern of the two models. Starting from the root node, if there is a node in the host that represent the same semantics (i.e., performance metric type), we remove the node from the second tree and set its equivalent node in the host as parent of its sub-trees.

Otherwise, we remove the node and its sub-trees from the second tree and merge them under the node's parent in the host. In this case, we also need to check if the node or its children represents a performance metric that is transfered from the host due to the model interaction (i.e., delegated performance metric), or vice versa. We draw a line pointing from the deputy model node to the delegator model node. Similarly at a node representing a composite metric, relevant nodes from the both models will connect to the node to reflect the cumulative performance effect. The merge continues until the second tree is empty. The algorithms for merging inference trees of model restructuring is presented in Appendix B. We will show an example of building inference tree for model restructuring in section 6.4.1.

In the section 6.4, we will illustrate how to generate performance knowledge about compositional models with two example applications, FLASH and PDLAHQR, using the approach presented above.

6.3 Hercule Support for Compositional Model Diagnosis

Hercule's singleton model analysis facilities can also support the compositional diagnosis if provided with the performance knowledge specific to the model composition. Given two models whose performance knowledge has been stored in the knowledge base, the user needs to generate and input the extra knowledge imposed by their interaction pattern to diagnose a specific algorithm, which includes the combined abstract event descriptions, composite metric evaluation rules, performance factors specific to the model interaction, and interfacing inference steps that link two inference trees together in accordance with their interaction pattern. The guidelines to generate the compositional knowledge from the already available base model knowledge have been provided in section 6.2. The knowledge engineering approach, in contrast to building everything from the scratch, can effectively reduce the users' burden enforced by the diagnostic process. In next section, we will apply Hercule to two real world applications to demonstrate the effectiveness of our diagnosis approach.

6.4 Experiments

6.4.1 ScaLAPACK Nonsymmetric QR algorithm – PDLAHQR

PDLAHQR is a parallel QR algorithm solving the nonsymmetric eigenvalue problem in ScaLAPACK. This implementation of the QR algorithm performs QR iterations implicitly by chasing multiple bulges down the subdiagonal of an upper Henssenberge matrix in parallel. The bulge chase dictates a variant of pipeline model that works as follows. The processors are arranged logically as a grid of R rows and C columns. The matrix is decomposed into $L \times L$ blocks, which are parceled out to the processors by a two-dimensional (block cyclic) torus wrap mapping. When the ith bulge is chased down to the $i + 1th$ row of processor, the $i + 1th$ bulge can start at the first row of processor. It is not until the i+1th bulge starts that the processors located on the $i + 1th$ row or the $i + 1th$ column are occupied and start to work. When the ith bulge reaches the $i + 1th$ row, first the leader of the $i + 1th$ row, usually the processor located on the grid diagonal, starts to do householder transform. And then the leader broadcasts the householder information horizontally to the $i + 1th$ row and vertically to the $i + 1th$ column so that the neighbor processors located at the same row or column as the leader become busy. This variant of pipeline model of parallelism increase scalability of the QR implementation significantly. Due to the block cyclic data distribution, neighbor processors communicate in two scenarios. The first is the leader processor broadcasting householder information to neighbor processors at the same row or column. The second happens when a bulge moves from one block to another, which may incur a neighbor communication of the border between the blocks that the two neighbors share. So we can view the application as a combination of pipeline and geometric decomposition model. The dynamic communication structure of four successive compute phases in the application is illustrated in the Figure 6.1, assuming a 3×3 processor grid.

Identify and describe model composition pattern

The first step towards generating performance knowledge is to investigate how the two singleton models, pipeline and geometric decomposition, interleave in the application.

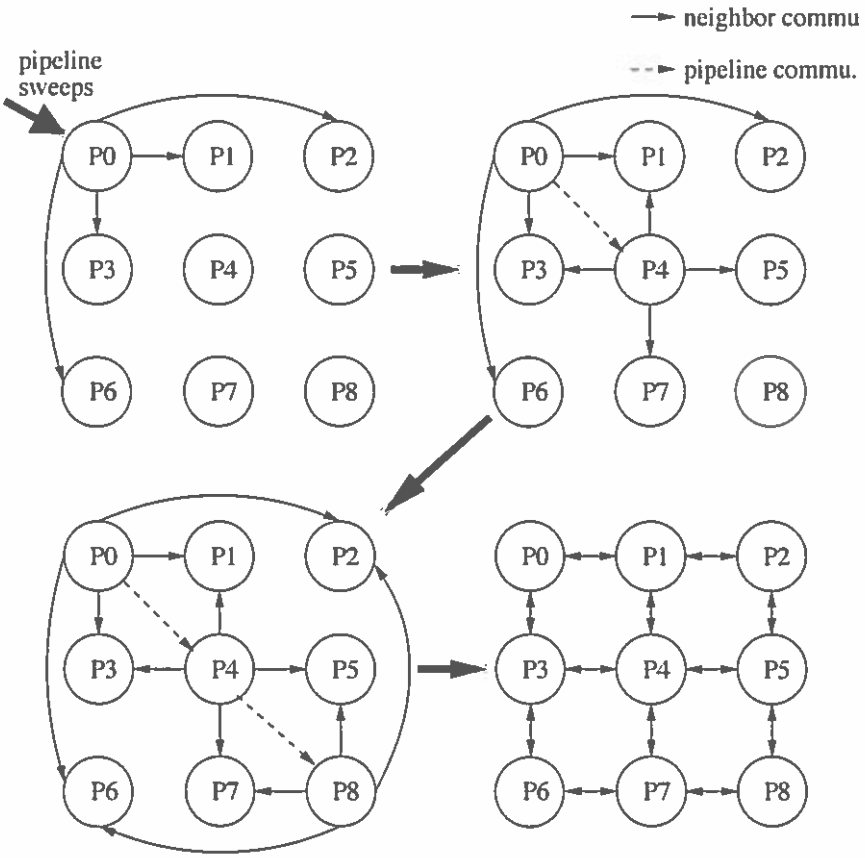


FIGURE 6.1: PDLAHQR dynamic communication structure in four successive compute phases on a 3x3 processor grid.

Generally pipeline and geometric decomposition behavior can be described as:

$$ITER_{pl}\{RECV_{pl} \circ COMPUTE_{pl} \circ SEND_{pl}\}$$

and

$$ITER_{gd}\{RECV^{*}_{gd} \circ COMPUTE_{gd} \circ SEND^{*}_{gd}\}$$

respectively, where $RECV_{pl}$ and $SEND_{pl}$ represent receive data from preceding pipeline stage and send data to succeeding stage, and $RECV_{gd}$ and $SEND_{gd}$ represent receive data from and send data to neighbors respectively. We use $ITER$ here instead of * to distinguish different iteration semantics in the two models.

Since there are two distinct types of processor, row leader and non-leader, we describe their behaviors in PDLAHQR separately as

$$\begin{aligned} &ITER_{pl}\{RECV_{pl} \circ COMPUTE_{pl} \circ SEND_{gd}^{row} \circ SEND_{pl}\} \circ \\ &ITER_{pl}\{SEND_{gd}^{col}\} \circ \\ &ITER_{gd}\{RECV_{gd}^{*} \circ COMPUTE_{gd} \circ SEND_{gd}^{*}\} \end{aligned} \tag{6.3}$$

for row leaders, and

$$\begin{aligned} &(ITER_{pl}\{RECV_{gd}^{row}\} | ITER_{pl}\{RECV_{gd}^{col}\}) \circ \\ &ITER_{gd}\{RECV_{gd}^{*} \circ COMPUTE_{gd} \circ SEND_{gd}^{*}\} \end{aligned} \tag{6.4}$$

for non-leaders. From the above behavior descriptions, we can see that the model composition in PDLAHQR falls into the category of model restructuring.

Performance metric characteristics

Based on the above model behavior descriptions, we shall focus on detecting and formulating delegated and composite performance delays distinct to the model restructuring. As seen in formula (6.4), since non-leader processors do not have explicit pipeline operations, a pipeline delay, such as idle time due to the pipeline start-up, handshaking, and wind-down, manifests as the communication delay at $RECV_{gd}^{row}$ or $RECV_{gd}^{col}$, which are classified as neighbor communication inefficiency in the original geometric decomposition model. There are also composite delays in the application. In the iteration $ITER_{gd}$, a row leader needs to exchange data with non-leader neighbor processors. $RECV_{gd}$ in the formula (6.3) may reflect not only work balance between the neighbors at $COMPUTE_{gd}$, but the extra more workload the row leader undertakes in two preceding $ITER_{pl}$ iterations than the non-leaders. So the communication delay occurring at the $RECV_{gd}$ should take into account pipeline compute's performance impact in its evaluation.

Merge inference trees of participant models

Constructing compositional model inference tree for PDLAHQR is simply to merge inference trees of pipeline and geometric decomposition. As shown in Figure 6.2, subtrees in the two models that share the same semantic parent node (e.g., communication and computation) combine into one single tree rooted at the parent node. Subtrees from different models remain independent unless they have nodes addressing a delegated or composite performance metric. *neighbor_comm.* node in geometric decomposition model, for instance, has *pl_fillup* and *pl_handshaking* nodes in pipeline model as its children. Connected with a delegation arrow, the relation means that an expensive neighbor communication may look for its cause at pipeline fill-up or handshaking. And compute imbalance, a commonly-seen performance phenomena in geometric decomposition, may be interpreted in part by the extensive pipeline compute considering the interleaving pattern of the two models. The relation is denoted by the composition arrow connecting the nodes from the two models.

Experiment results with PDLAHQR

We run the PDLAHQR program on a IBM pSeries 690 SMP cluster with 16 processors. We set a 4×2 processor grid and use PDLAHQR to compute the nonsymmetric eigenvalue of a 100×100 matrix. The Figure 6.3 shows performance profiles of the execution of PDLAHQR, where major program functions are presented in order of decreasing mean execution time across processors. We can see here that MPI_Bcast() and MPI_Recv() time dominate, which are main communication functions used in the pipeline and geometric decomposition model. Hercule first conducts an experiment to find a performance symptom, expensive communication cost.

```
dyna6-221:~/PerfDiagnosis lili$./model_diag PDLAHQR.clp
8pe.dup
-------------------------------------------------------
Found PDLAHQR.clp ... Loading
Begin diagnosing PDLAHQR program
... ...
Level 1 experiment -- collect performance profiles with
respect to computation and communication.
```

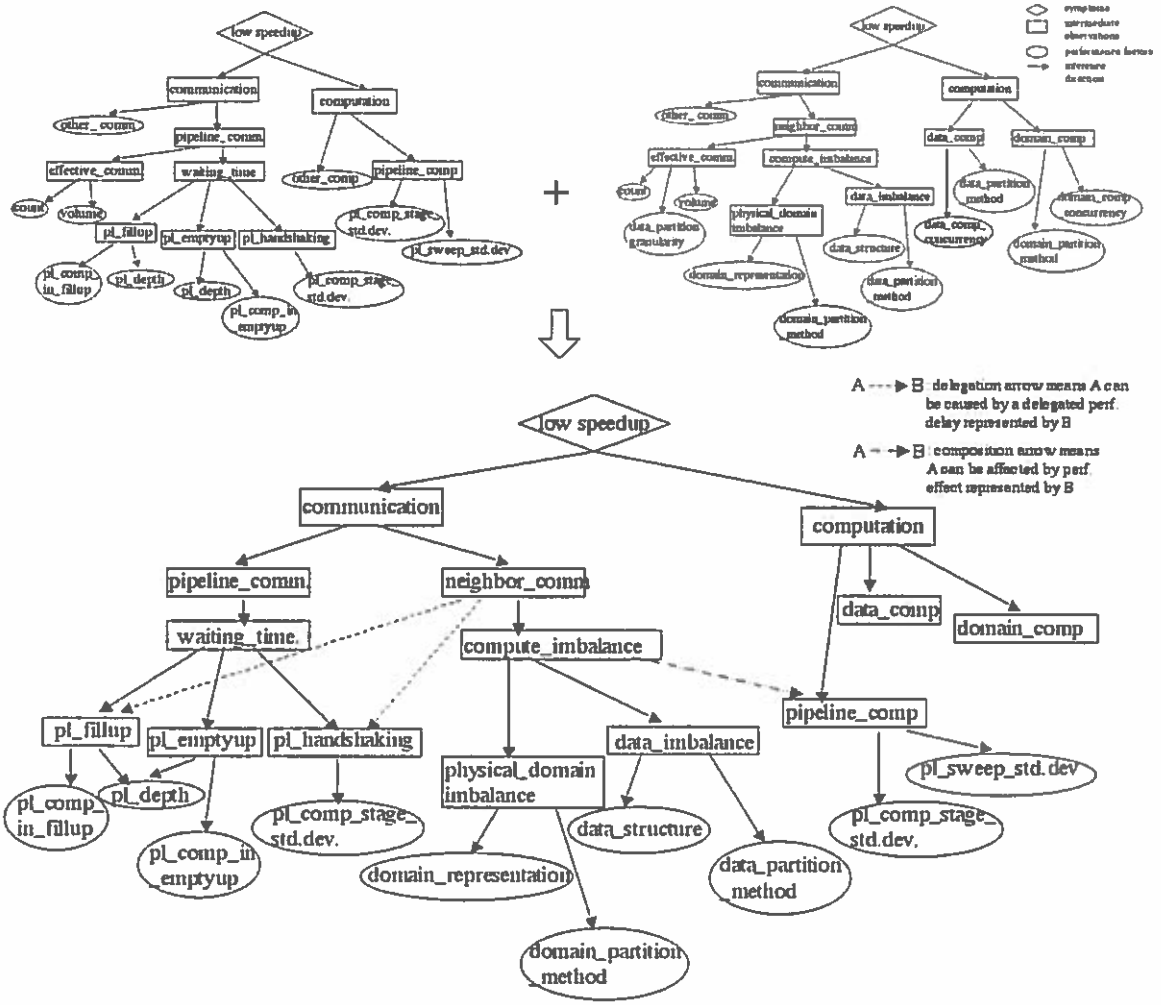


FIGURE 6.2: Construct compositional model inference tree for PDLAHQR. The top two trees represent pipeline and geometric-decomposition performance inference respectively, and they combine into the PDLAHQR inference tree on the bottom according to its model restructuring. Some subtrees are abbreviated in the composition model for conciseness.

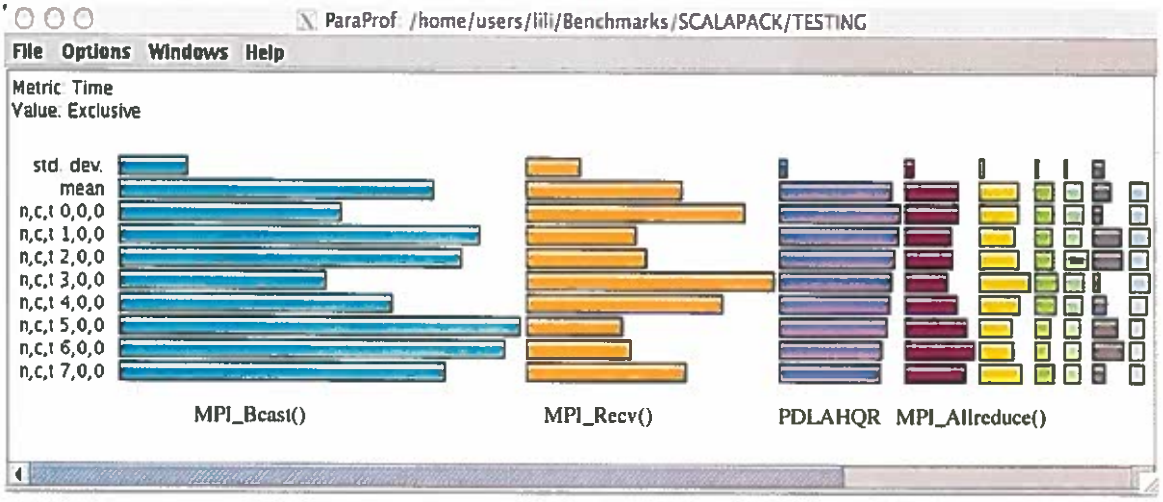


FIGURE 6.3: Paraprof view of performance profiles in the PDLAHQR run. We display most expensive program functions here.

```
do experiment ... ...

Average communication time account for 73.15% of overall
execution time. Communication cost dominates.
```

Hercule then investigates communication cost associated with models. It looks at geometric-decomposition performance first, and identifies the performance effects of the pipeline model.

```
Level 2 experiment -- collect performance profiles with
respect to two models used in the program, Pipeline and
Geometric-decomposition.
```

```
do experiment ... ...

Processors spent 13.23% of communication time in Pipeline
compute and 78.26% in Geometirc-decomposition compute.
```

```
First we look at geometric-decomposition compute.

Level 3 experiment for diagnosing geometric-decomposition
model related problems -- collect performance event trace
with respect to neighbor communication, and interaction cost
```

```
with pipeline model.
```

```
do experiment ... ...

Waiting time associatd with geometric-decomposition
comprises 37.32% of communication time.
```

```
In geometric-decomposition, load imbalance due to
interleaving with pipeline compute results in waiting
time that account for 17.83% of communications.
```

```
In geometric-decomposition, communication delay delegated
from pipeline comprises 10.55% of communication time.
```

```
Level 4 experiment for diagnosing pipeline impact on
geometric-decomposition performance -- collect performance
event trace with respect to pipeline fill-up, handshaking,
and wind-down impact on nearest neighbor communication.
```

```
do experiment ... ...

Pipeline fill-up delay propagates to compute/communciation
component in geometric-decomposition, which results in
waiting time that comprises 3.97% of communications.
```

```
Pipeline handshaking delay propagates to compute/commu.
component in geometric-decomposition, which results
in waiting time that comprises 6.58% of communications.
```

```
Pipeline wind-down delay propagates to compute/commu.
component in geometric-decomposition, which results in
waiting time that comprises 0.0 of communications.
```

Hercule then interpretes pipeline performance.

```
Level 3 experiment for diagnosing pipeline model related
problems -- collect performance event trace with respect to
pipeline compute, and interaction cost with geometric-
decomposition.
```

---

do experiment ... ...

Waiting time associatd with pipeline compute comprises 11.49% of communication time.

Pipeline waiting time due to the interaction with geometric-decomposition comprises 0.0 of communicaiton time.

Pipeline fill-up cost in explicit PL compute comprises 1.46% of communicaiton time. That is, processes at the pipeline tail are idle for 1.46% of communication time when pipeline is growing up.

Passing along data among successive pipeline stages (handshaking) in explicit PL compute comprises 5.98% of communication time.

Pipeline wind-down cost in explicit PL compute comprises 3.79% of communication time. That is, processes at pipeline head are idle 3.79% of communication time waiting for the pipeline to empty up at the end of the iteration.

Hercule is also able to explain performance of dynamic behaviors distinct to the PD-LAHQR algorithm. Since the algorithm is iterative the size of the active matrix is decreased by deflations. Each deflation causes a portion of the matrix to become inactive. As large portions of the matrix becomes inactive, processors begin to fall idle. Both pipeline and geometric-decomposition performance are affected by the dynamic program behavior.

Pipeline sweeps in some iterations do not grow deep enough to make every process active due to data characteristics of the iterations. Some processes therefore fall idle. Such idle time comprises 0.76% of communication time.

Dynamic load imbalance may arise as the compute step increases. The load imbalance causes processor idleness that comprises 6.65% of communication time in geometric-decomposition compute.

6.4.2 FLASH

FLASH [41, 47] is an astrophysical hydrodynamics code developed at the center for Astrophysical Thermonuclear Flashes at the University of Chicago. FLASH is intended for parallel simulations that solve the compressible Euler equations on an block-structured adaptive mesh. Adaptive Mesh Refinement (AMR) is handled using the PARAMESH library. PARAMESH employs a tree structure of logically Cartesian blocks to cover the computational domain. Each block in the domain is refined by halving the block along each dimension and generating a set of new sub-blocks, each of which has a resolution twice that of the parent block. When a block is de-refined, sibling blocks are removed. Each block is represented by a node in the tree structure. The node stores information about its parent block, child blocks, and neighboring blocks. An example of a two-dimensional domain and its tree structure is shown in Figure 6.4, adapted from [47]. For the sake of load-balance across processors, a redistribution of blocks is performed using a Morton space-filling curve [48] after all refinements and de-refinements are completed. So a block may be placed on a different processor from it parent or siblings.

Regardless of the particular algorithm, AMR dictates a set of basic operations which include guardcell filling, refining and de-refining grids, prolongation of the solution to newly created leaf blocks, restriction of the solution up the block tree, data redistribution when the grid block tree is rearranged (load balancing), etc. In the FLASH implementation, implied in the operations is the communications dictated by the grid block tree structure with the blocks being distributed to different processors and the maintenance of the tree structure with mesh refinement and de-refinement. We therefore view the FLASH code as a combination of two parallel computational models, AMR and Parallel Recursive Tree (PRT).

Identify and Describe Model Composition Pattern

AMR model consists of a set of basic mesh grid operations and data operations. The mesh grid operations includes:

- *AMR_Refinement*, refine a mesh grid

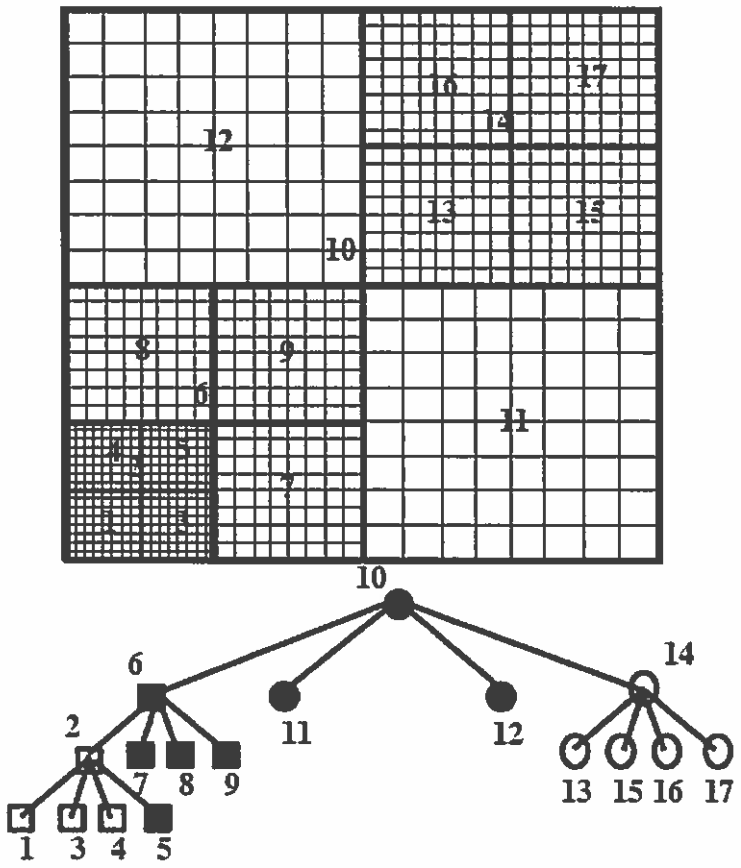


FIGURE 6.4: The tree structure that represents a set of blocks covering a fixed two-dimensional domain. A refined block has a cell size half that of the parent's. The number near the center of each block is its Morton number. The symbols in the tree shows on which processor the block is located on a four-processor machine.

- *AMR_Derefinement*, coarsen a mesh grid
- *AMR_LoadBalance*, even out work load among processors after a refinement or derefinement

The data operations corresponding to the mesh rebuilding includes

- *AMR_Guardcell*, update guard cells at the boundary of every grid block with data from the neighbors
- *AMR_Prolongation*, prolong the solution to newly created leaf blocks after refinement

- *AMR_Restriction*, restrict the solution up the block tree after derefinement
- *AMR_DataRedistribution*, data redistribution resulting from mesh redistribution when balancing workload

All the AMR operations in the Flash code are closely related to its grid block tree, which determines the communication needs and data movements. Parallel recursive tree model consists of a set of generic operations that include:

- *PRT_comm_to_parent*, communicate the processor on which the parent block is located.
- *PRT_comm_to_child*, communicate to the processor where the child block is located.
- *PRT_comm_to_sibling*, communicate to the processor where the sibling block is located.
- *PRT_build_tree*, initialize tree structure, or migrate part of the tree to another processor and rebuild the connection.

In Flash code, every AMR operation recalls the set of PRT operations to perform its function. The work mechanism of the $AMR\_Refinement$, for instance, is first to generate a list of children of the grid blocks to be refine, then to connect the new children blocks with off-processor neighbors designated by the parent blocks. The links between the new children and the parent neighbors are built through PRT operations. In other words, the computation of every component in the AMR model is reduced into PRT operations, while the semantic integrity of the two models are preserved. So the model composition in the FLASH code falls into the category of model nesting. The nesting forms a two-level model hierarchy where AMR is the root model that dictates the parallelism of the overall solution, and PRT the second-level model that addresses parallelization and implementation of the AMR operations.

Performance Modeling and Metric Characteristics

Due to the model nesting property of FLASH code, we intend to explain the performance in terms of its model hierarchy. The performance modeling starts with the root

model, then fills the lower-level model into the root framework to refine performance overhead categories[1]. Performance overhead types of the nested model composition is the union of the overheads associated with the individual participant models. To reflect the model hierarchy structure, however, we refine a performance overhead further into a number of context-aware types that indicate different occurrence circumstances within the model heirarchy. Guardcell filling, for instance, is one of the main sources of program inefficiency in AMR model. Due to the use of PRT model for data communication in the FLASH code, the guardcell filling overhead can be further broken down in terms of PRT overhead classes, i.e., communication cost with the parent nodes, child nodes, and neighbors nodes in the grid block tree. So it is possible to look into PRT performance as well within the AMR framework. Performance metrics that quantitatively evaluate the overheads are also organized in the hierarchy dictated by the model nesting to support the top-down performance problem search. In this way, we allow for capturing performance bugs at different levels of the model hierarchy and provide a context for performance interpretation in terms of the cross-level model interaction.

Merge Inference Trees of Participant Models

The construction of FLASH inference tree is essentially to extend the AMR inference tree with PRT inference steps. The Figure 6.5 shows the merge process. We can see that some AMR subtrees are extended with PRT branches, which means that a performance problem found relating to the subtree roots can be further tracked down to the PRT operations used. *parent_prolong* in AMR guardcell filling, for instance, involves communications to parent, child, and/or sibling in the grid tree. Its performance counts on these PRT operations along with the factors in the original AMR model. So when there is a problem with *parent_prolong*, we incorporate the relevant PRT inference steps to find the causes.

Experiment results with FLASH3

We experiment with the Sedov explosion simulation in the FLASH3.0, and run the simulation on a IBM pSeries 690 SMP cluster with eight processors. The Figure 6.6 shows

[1]For this work, we created new singleton models for AMR and PRT.

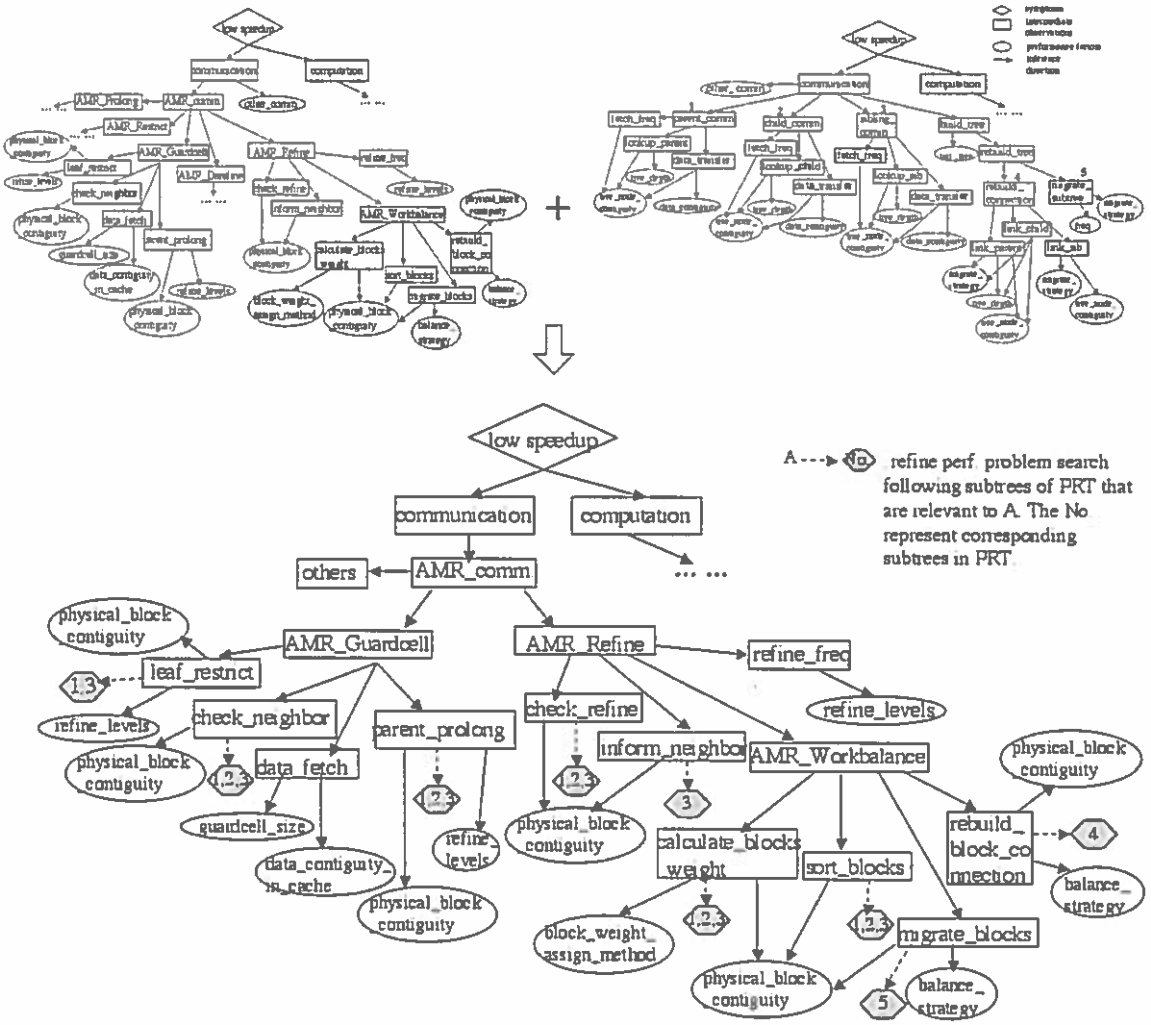


FIGURE 6.5: Construct compositional model inference tree for FLASH. The top two trees represent AMR and PRT performance inference respectively, and they combine into the FLASH inference tree on the bottom according to the model nesting in the FLASH code. Added PRT subtrees are highlighted in the FLASH tree and marked with indices in their original PRT tree. Some subtrees are abbreviated for conciseness.

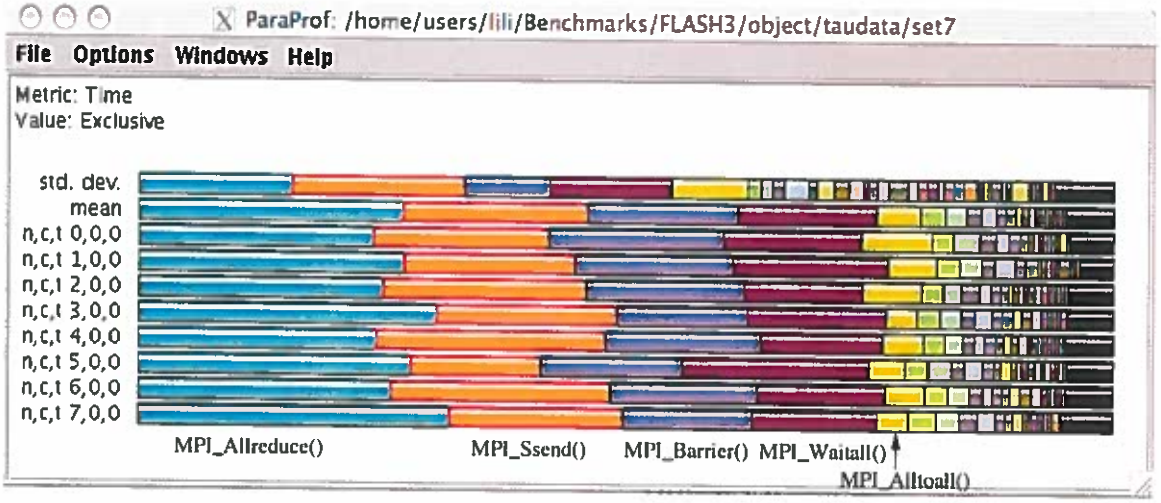


FIGURE 6.6: Paraprof view of performance profiles in the FLASH run. We display most expensive program functions here.

performance profiles of a simulation run, where major program functions on each processor are presented in order of decreasing mean exclusive execution time. From the profiles we can see that communications, including MPI_Ssend(), MPI_Allreduce(), MPI_Barrier(), MPI_Waitall(), dominate the runtime. But the profiles provide little insight into the performance of the AMR operations or the supporting data communications with PRT.

Hercule first conducts an experiment to find the performance symptom, expensive communication cost.

```
dyna6-221:~/PerfDiagnosis lili$./model_diag FLASH.clp
8pe.dup

_____________________________________________________________
Begin diagnosing AMR program
... ...
Level 1 experiment -- collect performance profiles with
respect to computation and communication.

_____________________________________________________________
do experiment 1... ...

Communication accounts for 80.70% of run time.
Communication cost of the run degrades performance.

_____________________________________________________________
```

Hercule then looks at the performance of the the top level model, AMR.

```
Level 2 experiment -- collect performance profiles with
respect to AMR refine, derefine, guardcell-fill, prolong,
and workload-balance.
```

```
do experiment 2... ...

Processes spent
4.35% of communication time in checking refinement,
2.22% in refinement,
13.83% in checking derefinement (coarsening),
1.43% in derefinement (coarsening),
49.44% in guardcell filling,
3.44% in prolongating data,
9.43% in dealing with work balancing,
```

Hercule then picks the most expensive AMR operation, guardcell-filling, to look at its performance in details, especially the performance of second level model PRT that implements guardcell-fillings.

```
Level 3 experiment for diagnosing grid guardcell-filling
related problems -- collect performance event trace with
respect to restriction, intra-level and inter-level
communication associated with the grid block tree.
```

```
do experiment 3... ...

Among the guardcell-filling communication, 53.01% is spent
restricting the solution up the block tree, 8.27% is spent
in building tree connections required by guardcell-filling
(updating the neighbor list in terms of morton order),
and 38.71% in transferring guardcell data among grid blocks.
```

```
The restriction communication time consists of 94.77%
in transferring physical data among grid blocks, and 5.23%
in building tree connections.
```

```
Among the restriction communication, 92.26% is spent in
collective communications.

Looking at the performance of data transfer in restrictions
from the PRT perspective,
remote fetch parent data comprises 0.0%,
remote fetch sibling comprises 0.0%,
and remote fetch child comprises 100%.
Improving block contiguity at the inter-level of the PRT
will reduce restriction data communication.
```

```
Among the guardcell-filling communication, 65.78% is
spent in collective communications.

Looking at the performance of guardcell data transfer from
the PRT perspective,
remote fetch parent data comprises 3.42%,
remote fetch sibling comprises 85.93%,
and remote fetch child comprises 10.64%.
Improving block contiguity at the intra-level of the PRT
will reduce guardcell data communication.
```

In FLASH, the AMR_Guarcell algorithm first restricts the data at "leaf" blocks up to the parent block, then all blocks that are leaf blocks or are parents of leaf blocks exchange guardcell data with any neighbor blocks they might have at the same refinement level. Hercule explains guardcell-filling performance from two dimensions here. It informs performance of each functional category, including AMR_Restriction, building tree connection, and guardcell data transportation. It also breaks down communications into collective and point-to-point (P2P) groups. The collective operations used in FLASH include MPI_Allreduce, MPI_Barrier, MPI_Alltoall, etc. The P2P includes MPI_Ssend, MPI_Irecv and MPI_Waitall pair, which are mostly used in the tree-related data transfer. From figure 6.6 we already know that these operations dominate the runtime. Hercule discriminates the performance of these two types of communication in AMR_Guardcell and AMR_Restriction, and interprets the P2P performance as reflected in the PRT compute. The users can thereby obtain an extensive insight into the FLASH performance from the perspective of the parallel models they coded with.

It is important to note is that Hercule automated all aspects of the diganosis process, including experiment construction, performance analysis, and performance causal inferencing.

6.5 Chapter Summary

Models of parallel computation are useful for discovering and explaining performance problems of parallel applications. For programs based on singleton models, we have shown that capturing knowledge of model behaviors, performance properties, and inference rules proves effective for diagnosis automation [49]. However, the approach will be limited in practice if we do not allow for more complex applications that combine multiple computational methods. In this chapter, we extend the model-based diagnosis methodology to support compositional models that integrate singleton computational patterns. Model nesting and model restructuring are two general compositional forms for which we discuss systematic steps to generate the performance knowledge necessary for automatic diagnosis of compositional programs. Our approach addresses the performance implications of model integration so that performance losses due to model interaction can be detected and interpreted. We implemented compositional performance diagnosis in Hercule framework and tested it with two scientific applications, FLASH and PDLAHQR. The experiment results reported here suggest that automatic diagnosis of compositional model performance is viable and effective.

Acknowledgements: The FLASH software used in this work was developed by the DOE-supported ASC/Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago.

CHAPTER VII

CONCLUSION

This dissertation presented a new approach to performance analysis of parallel programs, model-based automatic performance diagnosis. While previous research has produced a wide range of approaches for performance tuning, few of these ideas have sufficiently improved the ease of performance bug discovery and causal reasoning at a high-level program abstraction. Our view is that the primitive performance data evaluation and inferencing ought to be handled by performance tools, freeing the programmer to concentrate on the higher-level, algorithmic aspect. Our approach is motivated by the observation that by coding with computational models, parallel programs are already providing a well-defined abstraction of structural and parallelization design. We therefore devise a knowledge engineering approach that captures and codifies performance information from computational models. By using the performance knowledge, we can significant improve the quality of performance analysis as well as reduce the manual work imposed on the users compared to existing performance analysis tools. To substantiate this claim, we developed Hercule, a prototype automatic performance diagnosis tool, and apply it to a variety of parallel programs and scientific computations.

7.1 Research Contributions

We presented our model-based performance knowledge engineering approach, which addresses some important issues in performance analysis tool design.

- Our approach provides performance feedback at the program design-level, without requiring the users to evaluate and reasoning about performance with primitive measurements . The advantage of building a tool using our approach is that it can be used by novice programmers to find performance problems directly related to the programming model they coded with.

- Model-derived performance knowledge makes it possible to automate performance analysis processes. Model-specific information can guide experiment design, help pinpoint performance problems, and provide a semantic context for the problem-explanation. Performance experts implicitly use the information in analyzing their codes. We encode the expertise in a form that supports automatic performance diagnosis for non-expert users.

- Our model-based performance knowledge generation approach leverages a significant body of proven performance analysis techniques, such as behavioral modeling, performance modeling, metric formulation, and causal performance inferencing. We adapt and enhance the techniques by taking into account program semantics, and bring them together under a performance diagnosis framework.

- The knowledge engineering approach emphasizes adaptation to model variations which include algorithm-specific implementation and multiple model compositions. We provide a set of guidelines that help the user identify performance nuances introduced by the variations. Besides, each step towards the knowledge generation, like behavioral modeling and causal performance inferencing, is presented in a manner that can easily incorporate new performance information. In this sense, our approach produces a performance knowledge base that serves as a foundation of model-specific expertise which the users can inherit and extend into implementation-specific performance knowledge.

- We demonstrated the knowledge generation and encoding with both single and compositional models. And our later experiments with some real-world parallel applications show that the encoded knowledge, when judiciously interfacing to an inference

engine, can provide in-depth insight into parallel performance, while requiring only minimum intelligent input.

We developed the Hercule system that implements our model-based performance diagnosis approach. Hercule presents a new performance analysis framework with encoded performance knowledge being the core of the system. To date the knowledge base in Hercule system has included master-worker, pipeline, divide-and-conquer, AMR, and PRT model knowledge. We have demonstrated that Hercule is an viable system that is able to design performance experiments and conduct bug search and causal reasoning in an automated manner. Herule experiment results also corroborate the usefulness of designing tools based on our model-based approach. Experiments with Sweep3d application demonstrate Hercule's capability of relative performance diagnosis, i.e., contrasting performance of two problem scales and explaining the difference in terms of model semantics. Parallel applications representing different model composition styles are also investigated under Hercule framework. In this case, Hercule can not only provide model-specific performance feedbacks but identify the performance losses due to model interactions.

7.2 Future Research Directions

Our work represents the first step in exploiting models as a means to incorporate program semantics into performance analysis. Our results and experiences point to several directions for future research.

While we provide salient guidelines to derive performance knowledge from already available base model knowledge, thereby significantly reducing the complexity of knowledge engineering for compositional models, the process is still manual. As more singleton and compositional models are developed, the practice will improve in quality and more reuse will be possible. An interesting area for future work is to consider automatic techniques to transform and merge existing singleton performance knowledge into performance knowledge according to compositional rules.

There are many useful features that we could include in future versions of Hercule. We will continue to enhance Hercule adaptability to the nuances and complexities of parallel programs. In current version of Hercule, the user needs to identify algorithmic variations to the model stored in knowledge bases and encode the differences into performance knowledge necessary for automatic diagnosis. Hercule may help tackle the complexity by supporting for knowledge customization, e.g., specify differences in abstract event descriptions and metric evaluation rules. Hercule then transforms the differences into system-readible format automatically and incorporates them into relevant diagnosis system components.

Hercule may also target irregular, non-model based parallel programs. Instead of exhaustive performance problem diagnosis, the goal here is to find and explain major performance issues by loosely matching the program with models. As more models are developed, we envision that a wide variety of parallel execution patterns will be captured and encoded in abstract event library. Hercule may allow the user to look up close behavioral descriptions in abstract event library, which may be from different models, and specifies the combination forms of the retrieved abstract events. Hercule then automatically composes the performance inferencing processes associated with these abstract events into performance knowledge specific to the program. There will certainly be a diagnosis validity testing step involved afterward.

Although Hercule already addresses scalability with selective instrumentation and multiple level experiment management, the performance data processing will be bottleneck of causal inferencing as problem scale increases. Hercule may include proven scalable performance data management techniques under its framework. PerfExplorer [50], for instance, is such a technique that uses data mining operations to simplify the management and analysis of large-scale parallel performance profiles. Interfacing to the techniques can improve Hercule efficiency in the early inferencing steps that involve identifying a predominant performance phenomenon to focus on and synthesizing a large amount of measurement dataset into performance metrics at a higher level of program abstraction.

APPENDIX A

MERGING INFERENCE TREES FOR MODEL NESTING

Algorithm 1 Algorithms for merging inference trees in model nesting

Algorithm MERGE(T^{root}, T^{child})
Input:
T^{root} - inference tree of root model
T^{child} - inference tree of child model
Output:
T^{root} - root model inference tree extended with child model inference steps
BEGIN
ROOT $\Leftarrow$ root node of T^{root}
for all child node N_i of ROOT **do**
look at performance metric M_i the node N_i represents
if M_i involves a component which is implemented by model T^{child} **then**
if all children of N_i are leaf nodes **then**
BRANCHES $\Leftarrow$ SEARCH($T^{root}_{N_i}$, T^{child})
add BRANCHES as subtree of N_i
else
MERGE($T^{root}_{N_i}$, T^{child})
end if
end if
end for
END

Algorithm SEARCH(T_N^{root}, T^{child})
Input:
  T_N^{root} - a subtree of root model rooted at node N
  T^{child} - inference tree of child model
Output:
  subtrees of T^{child} that refines N inference
BEGIN
SUBTREES $\Leftarrow \phi$
ROOT $\Leftarrow$ root node of T^{child}
if N refers to all children of ROOT **then**
  SUBTREES $\Leftarrow T^{child}$
else
  for all child node C_i of ROOT **do**
    SUBTREES $\Leftarrow$ SUBTREES $\cup$ SEARCH(T_N^{root}, $T_{C_i}^{child}$)
  end for
end if
return SUBTREES
END

APPENDIX B

MERGING INFERENCE TREES FOR MODEL RESTRUCTURING

Algorithm 2 Algorithms for merging inference trees in model restructuring

Algorithm MERGE(T^{host}, T^{2nd})
Input:
T^{host} - inference tree of host model
T^{2nd} - another inference tree that is to be merged into host
Output:
T^{host} - host model inference tree extended with 2nd model inference steps
BEGIN
ROOT $\Leftarrow$ root node of T^{2nd}
PARENT $\Leftarrow$ LOOKUP(ROOT, T^{host}) {look up a node in T^{host} that has the same semantics as ROOT}
if PARENT $\neq$ NULL **then**
  remove ROOT from T^{2nd}
  for all subtrees $T^{2nd}_{C_i}$ of ROOT **do**
    set PARENT as $T^{2nd}_{C_i}$s parent node in host
    MERGE(T^{host}_{PARENT}, $T^{2nd}_{C_i}$) {T^{host}_{PARENT} is a subtree in T^{host} that is rooted at PARENT}
  end for
else
  if T^{2nd} has parent in host **then**
    PARENT $\Leftarrow$ T^{2nd}s parent in host
    add T^{2nd} as a subtree of PARENT
    LINK_INTERACT(T^{host}_{PARENT}, T^{2nd}) {add branches due to model interaction}
  end if
end if
END

Algorithm LOOKUP(N, T^{host})
Input:
N - a node to be merged into host
T^{host} - inference tree of host model
Output:
The node in T^{host} that has the same semantics as N
BEGIN
ROOT $\Leftarrow$ root node of T^{host}
if ROOT and N represent the same performance metric **then**
return ROOT
else
let $T^{host}_{C_i}(i = 1, \ldots, k)$ be subtrees of ROOT
$i \Leftarrow 1$
while $i <= k$ **do**
NODE $\Leftarrow$ LOOKUP(N, $T^{host}_{C_i}$)
if NODE == NULL **then**
$i++$
else
return NODE
end if
end while
return NULL
end if
END

Algorithm LINK_INTERACT(T^{host}, T^{2nd})
Input:
T^{host} - inference tree of host model
T^{2nd} - inference tree of another model to be merged
Output:
T^{host} - host model inference tree extended with branches due to interaction with the 2nd model
BEGIN
ROOT $\Leftarrow$ root node of T^{2nd}
NODE_INTERACT(T^{host}, ROOT)
for all subtrees $T^{2nd}_{C_i}$ of ROOT **do**
LINK_INTERACT(T^{host}, $T^{2nd}_{C_i}$)
end for
END

Algorithm NODE_INTERACT(T^{host}, NODE)
Input:
T^{host} - inference tree of host model
NODE - a node to be merged into host
Output:
T^{host} - host model inference tree extended with branches due to interaction with NODE in the 2nd model
BEGIN
ROOT $\Leftarrow$ root node of T^{host}
if performance metric at ROOT transforms into the one at NODE **then**
draw a delegation arrow connecting NODE to ROOT
else if metric at NODE transforms into the one at ROOT **then**
draw a delegation arrow connecting ROOT to NODE
end if
if metric at ROOT contributes to metric at NODE **then**
draw a composition arrow connecting NODE to ROOT
else if metric at NODE contributes to metric at ROOT **then**
draw a composition arrow connecting ROOT to NODE
end if
for all subtrees $T_{C_i}^{host}$ of ROOT **do**
NODE_INTERACT($T_{C_i}^{host}$, NODE)
end for
END

BIBLIOGRAPHY

[1] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *IEEE Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[2] J. C. Yan, "Performance tuning with AIMS – an Automated Instrumentation and Monitoring System for multicomputers," in *proceedings of 27th Hawaii International Conference on System Sciences*, pp. 625–633, 1994.

[3] University of Illinois at Urbana-Champaign, Department of Computer Science, "Sv-Pablo." [Online]. Available: http://www.renci.org/projects/pablo.php.

[4] University of Oregon, Department of Computer and Information Science, "TAU User's Guide." [Online]. Available: http://www.cs.uoregon.edu/research/tau/.

[5] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, ch. Analytical Modeling of Parallel Programs, pp. 195–231. Addison-Wesley, Harlow, England, 2003.

[6] T. Fahringer and C. S. Jr., "Aksum: a performance analysis tool for parallel and distributed applications," in *Performance analysis and grid computing*, pp. 189 – 208, Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[7] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent, "Hpcview: A tool for top-down analysis of node performance," *The Journal of Supercomputing*, vol. 23, pp. 81–104, 2002.

[8] B. L. Massingill, T. G. Mattson, and B. A. Sanders, "Some algorithm structure and support patterns for parallel application programs," in *proceeding of 9th Pattern Languages of Programs Workshop*, [Online]. Available:http://jerry.cs.uiuc.edu/ plop/plop2002/proceedings.html, 2002.

[9] B. L. Massingill, T. G. Mattson, and B. A. Sanders, "Patterns for parallel application programs," in *proceedings of 6th Pattern Languages of Programs Workshop*, [Online]. Available:http://www.cise.ufl.edu/research/ParallelPatterns/. 1999.

[10] N. Carriero and D. Gelernter, "How to write parallel programs: A guide to perplexed," *ACM Computing Surveys*, vol. 21, no. 3, pp. 323–357, 1989.

[11] Lawrence Livermore National Laboratory, "The ASCI sweep3D benchmark." [Online]. Available: http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/.

[12] Lawrence Livermore National Laboratory, "The ASCI Sphot Benchmark." [Online]. Available: http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks/limited/sphot/.

[13] T. Fahringer and C. S. Jr., "Modeling and detecting performance problems for distributed and parallel programs with javapsl," in *proceedings of Supercomputing (CD-ROM)*, 2001.

[14] APART Esprit IV Working Group on Automatic Performance Analysis: Resources and Tools. [Online]. Available: http://www.fz-juelich.de/zam/RD/coop/apart/.

[15] M. E. Crovella and T. J. LeBlanc, "Performance debugging using parallel performance predicates,," in *proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 140 – 150, 1993.

[16] T. Suzuoka, J. Subhlok, and T. Gross, "Performance debugging based on scalability analysis," tech. rep., Carnegie Mellon University, 1994.

[17] Forschungszentrum Jülich, "EXPERT - Extensible Performance Tool." [Online]. Available: http://www.fz-juelich.de/zam/kojak/.

[18] W. M. Jr., *Parallel Performance Understanding via Integration of Modeling and Diagnosis*. PhD thesis, University of Rochester, 1996.

[19] J. W. Meira, T. J. LeBlanc, and V. A. F. Almeida, "Using cause-effect analysis to understand the performance of distributed programs," in *proceedings of SIGMETRICS symposium on Parallel and distributed tools*, pp. 101–111, 1998.

[20] J. Kohn and W. Willams, "ATExpert," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 205–222, 1993.

[21] E. Cesar, J. Mesa, J. Sorribes, and E.Luque, "Modeling master-worker applications in poetries," in *proceedings of 9th International Workeshop on High-Level Parallel Programming Models and Supportive Environments*, pp. 22–30, 2004.

[22] R. Rajamony, *Prescriptive performance tuning: the R_X approach*. PhD thesis, Rice University, 1998.

[23] R. Rajamony and A. L. Cox, "Performance debugging shared memory parallel programs using run-time dependence analysis," in *proceedings of SIGMETRICS*, pp. 75 – 87, 1997.

[24] S. V. Adve, "Using information from the programmer to implement system optimizations without violating sequential consistency,," tech. rep., Rice University, 1996.

[25] A. Espinosa, *Automatic Performance Analysis of Parallel Programs*. PhD thesis, University Autonoma de Barcelona, Barcelona, Spain, 2000.

[26] B. R. Helm, A. D. Malony, and S. F. Fickas, "Capturing and automating performance diagnosis: the poirot approach," tech. rep., University of Oregon, 1993.

[27] A. D. Malony and B. R. Helm, "A theory and architecture for automating performance diagnosis," *Future Generation Computer Systems*, vol. 18, pp. 189–200, 2001.

[28] P. C. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," *ACM Trans. on Computer Systems*, vol. 13, no. 1, pp. 1–31, 1995.

[29] G. Blelloch and G. Narlikar, "A practical comparison of N-body algorithms," *Parallel Algorithms. Series in Discrete Mathematics and Theoretical Computer Science*, vol. 30, pp. 81–96, 1997.

[30] F. Aurenhammer, "Voronoi diagrams - a survey of a fundamental geometric data structure," *ACM Comput. Surv.*, vol. 23, no. 3, pp. 345–405, 1991.

[31] J. C. Giarratano and G. D. Riley, *Expert Systems: Principles and Programming*. Course Technology, Boston, MA, USA, 1998.

[32] Public domain software, "CLIPS: A Tool for Building Expert Systems." [Online]. Available: http://www.ghg.net/clips/CLIPS.html.

[33] I.-C. Wu and H. T. Kung, "Communication complexity for parallel divide-and-conquer," in *proceedings of the 32nd annual symposium on Foundations of computer science*, pp. 151 – 162, 1991.

[34] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[35] P. Brucker, C. Dhaenens-Flipo, S. Knust, S. Kravchenko, and F. Werner, "Complexity results for parallel machine problems with a single server," *Journal of Scheduling*, vol. 5, no. 6, pp. 429–457, 2002.

[36] Pallas, "VAMPIR - Visualization and Analysis of MPI Resources." [Online]. Available: http://www.llnl.gov/icc/lc/DEG/vgv.html.

[37] R. Bell, A. D. Malony, and S. Shende, "Paraprof: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis," in *EUROPAR*, vol. 2790 of *Lecture Notes in Computer Science*, pp. 17–26, Springer, 2003.

[38] D. Abramson, I. Foster, J. Michalakes, and R. Sosic, "Relative debugging: a new paradigm for debugging scientific applications," *The Communications of the Association for Computing Machinery*, vol. 39, no. 11, pp. 67–77, 1996.

[39] K. Karavanic and B. Miller, "A framework for multi-execution performance tuning," in *On-line Monitoring Systems and Computer Tool Interoperability* (T. Ludwig and B. Miller, eds.), pp. 61–89, Nova Science Publishers, Inc., 2004.

[40] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore, "An algebra for cross-experiment performance analysis," in *proceedings of ICPP*, pp. 63 – 72, 2004.

[41] ASC/Alliances Center for Astrophysical Thermonuclear Flashes, "FLASH3.0." [Online]. Available: http://flash.uchicago.edu/website/home/.

[42] G. Henry, D. Watkins, and J. Dongarra, "A parallel implementation of the nonsymmetric qr algorithm for distributed memory architectures," *SIAM Journal on Scientific Computing*, vol. 24, no. 1, pp. 284–311, 2002.

[43] S. MacDonald, D. Szafron, and J. Schaeffer, "Rethinking the pipeline as object-oriented states with transformations," in *proceedings of 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pp. 12–21, 2004.

[44] P. R. Amestoy, I. S. Duff, J. Koster, and J. L'Excellent, "A fully asynchronous multifrontal solver using distributed dynamic scheduling," *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 1, pp. 15–41, 2002.

[45] L. Li and A. D. Malony, "Model-based performance diagnosis of master-worker parallel computations," in *proceedings of Europar2006*, vol. 4128 of *Lecture Notes in Computer Science*, pp. 35–46, Springer, 2006.

[46] L. Li, A. D. Malony, and K. Huck, "Model-based relative performance diagnosis of wavefront parallel computations," in *High Performance Computing and Communications*, vol. 4192 of *Lecture Notes in Computer Science*, pp. 200–209, Springer Berlin/Heidelberg, 2006.

[47] A. C. Calder, B. C. Curtis, L. J. Dursi, B. Fryxell, G. Henry, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. X. Timmes, H. M. Tufo, J. W. Turan, and M. Zingale, "High performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors," in *procceedings of Supercomputing (CD-ROM)*, 2000.

[48] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *proceedings of Supercomputing*, pp. 12–21, 1993.

[49] L. Li and A. D. Malony, "Knowledge engineering for automatic parallel performance diagnosis," *To appear in Concurrency and Computation: Practice and Experience*, 2007.

[50] K. A. Huck and A. D. Malony, "Perfexplorer: A performance data mining framework for large-scale parallel computing," in *proceedings of Supercomputing (CD-ROM)*, 2005.