Model-Based Performance Diagnosis of Master-Worker Parallel Computations

Li Li and Allen D. Malony

Performance Research Laboratory
Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
{lili,malony}@cs.uoregon.edu

Abstract. Parallel performance tuning naturally involves a diagnosis process to locate and explain sources of program inefficiency. Proposed is an approach that exploits parallel computation patterns (models) for diagnosis discovery. Knowledge of performance problems and inference rules for hypothesis search are engineered from model semantics and analysis expertise. In this manner, the performance diagnosis process can be automated as well as adapted for parallel model variations. We demonstrate the implementation of model-based performance diagnosis on the classic Master-Worker pattern. Our results suggest that pattern-based performance knowledge can provide effective guidance for locating and explaining performance bugs at a high level of program abstraction.

Keywords: Performance diagnosis, parallel models, master-worker, measurement, analysis.

1 Introduction

Performance tuning (a.k.a. *performance debugging*) is a process that attempts to find and to 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. Implicit in this process is the expert's knowledge of the program's code structure, its parallelism approach, and the relationship of application parameters with performance factors. 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*). This paper focuses on *parallel performance diagnosis* and how it can be supported as an automated knowledge-based process in performance analysis tools.

Performance diagnosis, as a process, is best based on understanding of how expert parallel programmers debug performance problems. That is, we should regard performance diagnosis as an intelligent system wherein we capture *knowledge* about performance problems and how to detect them, and then apply this knowledge in a diagnosis framework. The key idea is to extract performance knowledge from parallel computational models that represent structural and communication pattern of a program. The models provide semantically rich descriptions that enable better interpretation and understanding of performance. The goal is to engineer the performance knowledge to support bottom-up inference of performance causes effectively. A diagnosis system can then use the performance knowledge for performance problem search and reasoning. The problem we focus on in this paper is the knowledge engineering required for model-based performance diagnosis. We will show in a particular scenario, the classic *Master-Worker* parallel model, that the performance knowledge derived from parallel models provides a sound basis for automating performance diagnosis processes and can explain performance loss from high-level computation semantics.

In Section §2, we more formally discuss performance diagnosis as a general intelligent process and provide background on why we advocate a model-based diagnosis approach. From this perspective, Section §3 describes our approach to engineering performance knowledge and problem inference. A prototype diagnosis system, *Hercule*, was developed based on this approach and is presented. The Master-Worker pattern is illustrated in section §4 to demonstrate how performance knowledge is engineered in Hercule and to show automatic performance diagnosis in action. Section §5 highlights related research and Section §6 concludes with observations and future work.

2 Model-based Performance Diagnosis

Performance diagnosis is the process of locating and explaining sources of performance loss in a parallel execution. Expert parallel programmers often improve program performance by iteratively running their programs on a parallel computer, then interpreting the experiment results and performance measurements to suggest changes to the program. Specifically, the process involves:

Designing and running performance experiments. Parallel computing researchers have developed integrated measurement systems to facilitate performance analysis [18, 15, 2]. The performance experiments specify input data, number of processors, and other parameters. The experiments also decide on points of instrumentation and what 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, parallelization overhead, speedup, efficiency, and cost. By comparing the metric values computed from performance data with what is expected, we can find symptoms such as low scalability, poor efficiency, and high overhead.

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 [9], such as load balance and communication cost, or tracking down specific source code fragments that are responsible for major performance loss. Performance analysis expertise and knowledge about code structure and parallelization design can help form performance hypotheses, capture supporting performance information, synthesize raw performance data to test the hypotheses, and iteratively refine hypotheses toward higher-level abstractions until some cause is found.

A parallel computational model [13, 14], also called a parallel pattern [16] or programming paradigm [6] in the literature, is a recurring parallel solution to a class of problems. Typical models include master-worker, pipeline, divide-and-conquer, and domain decomposition [13]. Models usually describes computational components and their behaviors (semantics) and how multiple threads of execution interact and collaborate in a parallel solution (parallelism). Parallel programming 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 realistic program implementations. Specifically, we envision that computational models can play an active role in the following aspects of performance diagnosis:

Selective instrumentation. 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 experiment to generate the information. Parallel models identify major computational components in a program, and can therefore guide the code instrumentation and 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 associated behaviors.

Expert analysis of performance problems. There is a collection of commonly-used parallel models for constructing parallel applications. Expert knowledge about the model performance includes typical performance properties and corresponding factors at the level of program/algorithm design. If we can represent and manage the performance knowledge in a proper manner, they will effectively drive diagnosis process with little or no user intervention.

3 Performance Diagnosis Engineering and Hercule

To build a performance diagnosis system, we need to generate performance knowledge from computational models and represent it in a knowledge base for

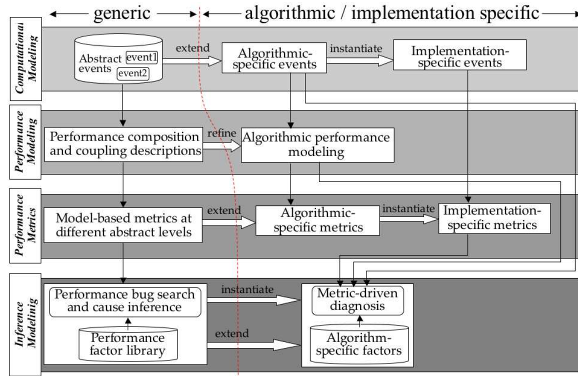

Fig. 1: Generating performance knowledge from models. The dashed line draws a boundary between model-based and algorithm/implementation specific knowledge generation.

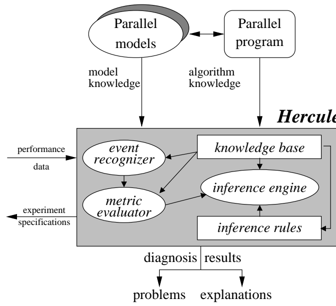

Fig. 2: Hercule Framework

use in experimentation and problem discovery. Extracting performance knowledge from parallel computational models involves four types of actions, which are shown in Figure 1. The *computational modeling* captures knowledge of program execution semantics as behavioral patterns 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 analysis, and diagnosis inferencing.

Performance modeling is carried out based on structural information in the abstract events. The modeling identifies performance attributes with respect to the behavior patterns represented by abstract events and model-specific performance overhead categories. *Performance metrics* are then defined, in terms of performance attributes in related abstract events, to evaluate the performance properties for problem interpretation. *Inference modeling* (i.e., performance bug search) is driven by the metric and property evaluation. Cause inference tries to explain found performance problems with performance-critical program design factors, as specific to the particular computational model. The performance problem analysis and cause refinement are captured in the form of an inference tree linking symptoms to sources.

Algorithmic implementations of a computational model may introduce new performance knowledge with regard to behavioral models, performance properties, performance-critical design factors, or cause inference. Following our four-step 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 on the right hand part of Figure 1. In our design and implementation of a model-based performance diagnosis system, we will allow the expression of al-

gorithmic features and incorporating it into the inference system that is initially based on generic model knowledge.

We have built a prototype automatic performance diagnosis system called *Hercule*[1], which implements the model-based performance diagnosis approach discussed above; see Figure 2. The Hercule system operates as an expert system within a parallel performance measurement and analysis toolkit, in this case, the TAU [2] performance system. Hercule includes a knowledge base composed of an abstract event library, metrics set, and performance factors for individual parallel models. Below, we describe in more detail how the performance diagnosis engineering is accomplished in Hercule.

The *abstract event description* used in EBBA [4] is adapted in Hercule to describe behavioral characteristics of a target computational model. The 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. 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., interprocess communication or regular routine invocations), or an instance of other abstract event type. The *expression* is a specification that names the constituent events and enforces their occurrence order using event operators. The order can be *sequential* ($\circ$), *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. *Constraints* indicate what attribute values an instance of an abstract event type must possess to match its corresponding expression members and associated events. *Performance attributes* present performance properties of the behavior model an abstract event type represents and computing rules to evaluate them. Figure 3 in the next section shows an example abstract event for the Master-Worker (M-W) computational model.

Hercule implements the abstract event representation in a Java class library. The *event recognizer* in Hercule fits event instances into abstract event descriptions as performance data stream flows through it. It then feeds the event instances into Hercule's performance model evaluator - *metric evaluator*. Performance models in Hercule are coded as Java classes used to represent model-specific metrics and associated performance formulations. The performance metrics will be evaluated based on the related abstract event instances. The event recognizer and metric evaluator can incorporate algorithm-specific abstract event definitions and metric computing rules.

[1] The name was chosen in the spirit of our earlier performance diagnosis project, *Poirot* [12].

Perhaps the most interesting part of the Hercule knowledge engineering is the cause inferencing system. The expert knowledge used to reason about performance problems based on model symptoms can be 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 encode inference trees with production rules. A production 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 [1] expert system building tool to describe production rules, and the CLIPS inference engine for operation. The inference engine provided in CLIPS 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. Due to the limitation of space, we refer readers to elsewhere [11] for details of encoding performance knowledge with CLIPS.

The effort involved in implementing performance knowledge base for a computational model consists of two parts: acquiring knowledge with the approach presented above and encoding the knowledge with abstract event specification, performance formulation, and production rules. 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 developer 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 Master-Worker Parallel Pattern and Diagnosis

A widely used parallel computation pattern is the classic Master-Worker (M-W) model. Here we use the M-W model to demonstrate the performance diagnosis methodology above and show how it is implemented in the Hercule framework. 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. The master usually employs certain task scheduling algorithms to achieve load balance and minimize workspan. 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.*
- *Number and complexity of tasks assigned to the workers.*
- *Task setup costs in the master and the task scheduling method.*
- *Number of worker processors.*

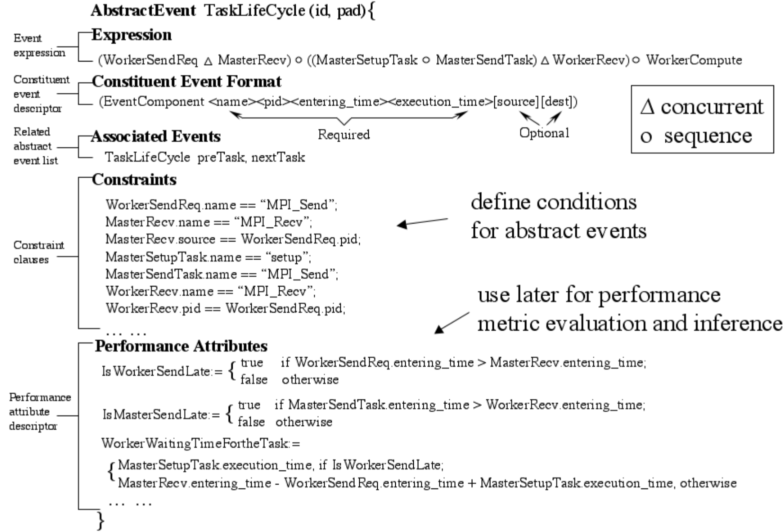

Fig. 3: An abstract event description of Master-Worker model

– *Task scheduling strategy*

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 3.

Given the program behavior, we can formulate M-W performance models. For instance, 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):

$$t_{worker} = t_{init} + t_{comp} + t_{comm} + t_{wait} + t_{final} \tag{1}$$

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.

Performance coupling of a worker with the master and the rest of peer workers manifests four performance overheads – t_{seq} (the master initialization and

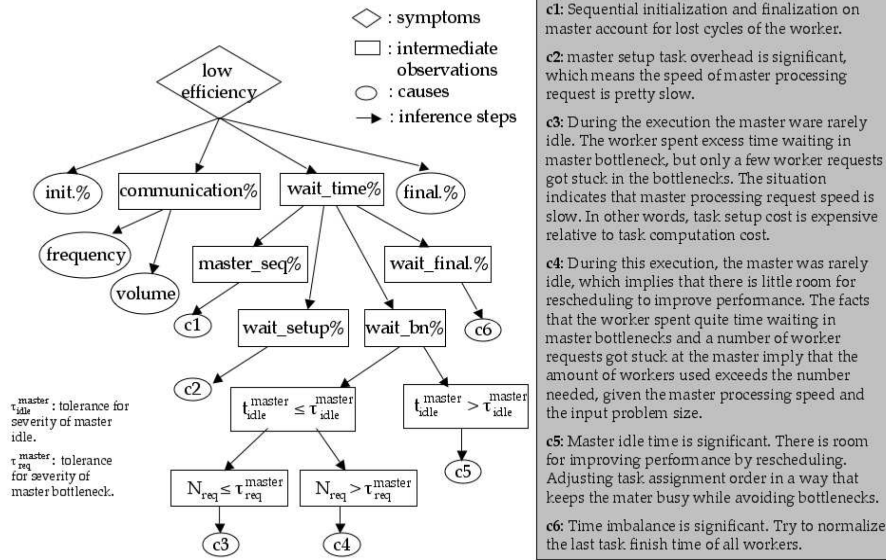

Fig. 4: Inference Tree for Performance Diagnosis of M-W programs.

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).

$$t_{wait} = t_{seq} + t_{w-setup} + t_{w-bn} + t_{w-final} \tag{2}$$

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.

$$\textbf{worker efficiency} := \frac{t_{comp}^{worker}}{t_{worker}} \tag{3}$$

Refining each item in model (2), we obtain metrics of worker wait time:

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

where M is the number of tasks the worker processes altogether, t_{setup}^{i} the amount of time for setting up task i, t_{w-bn}^{i} is the waiting time due to master bottleneck when requesting the ith task, t_{wait}^{i} is the total amount of worker idle

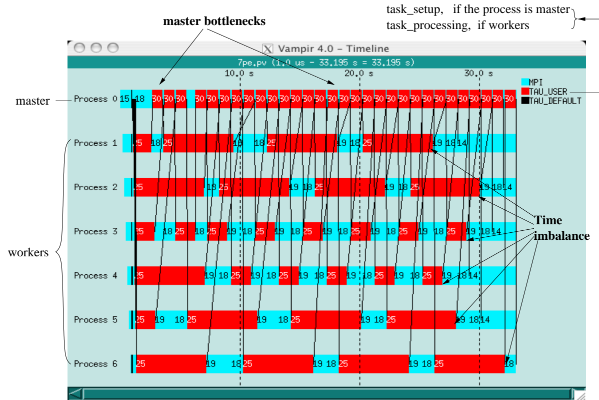

Fig. 5: Vampir timeline view of an example M-W program execution.

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

Fig. 6: Metric values of the run.

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.

Now we can incorporate these performance factors and metrics in diagnosis inference rules. An inference tree is created for every symptom. The inference tree for explaining low efficiency of a worker process, for instance, is shown in Figure 4. The root is the symptom to be diagnosed, the branch nodes are intermediate observations obtained so far (i.e., a performance evaluation with respect to a performance metric, such as waiting time is a *significant* percentage of total elapsed time), and need further performance evidences to explain, and the leaf nodes are an explanation of the root symptom in terms of high-level performance factors. It is interesting to note that nodes at different inference tree levels may enforce varying experiment specifications. Our diagnosis system can construct the experiments according to the abstract event descriptions from which the metrics derive.

We tested Hercule's performance diagnosis capability for the M-W pattern 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. The M-W synthetic program was implemented using MPI.

For the results discussed below, we introduced in the M-W program a performance fault targeting the impact of master-request-processing speed on overall performance. Figure 5 presents a Vampir [3] timeline view of a parallel execution with one master and six workers. The event trace and profiles are generated by TAU [2] with only major model components being instrumented. The red regions in the figure represent task setup periods at the master and task processing periods at the workers. Light blue regions represent MPI function calls. Note that

```
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...
```

Fig. 7: Diagnosis result output from Hercule of the M-W test program.

blocking (waiting) time of processors is implicitly included in the elapsed time of blocked **MPI_Send**, **MPI_Recv** and **MPI_Finalize** operations.

Given the program and performance knowledge associated with M-W model, Hercule will automatically request three experiments during the diagnosis of this problem. The inference process and diagnosis results of these experiments are presented in Figure 7. 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 second experiment to evaluate individual overheads in equation (1). Waiting time cost stands out as a result of this inference step. The third experiment then targets performance loss categories in equation (2). Figure 6 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 Related Research

There are several related projects to our work. Paradyn [15] is a performance analysis system that automatically locates bottlenecks using the W^3 search

model. According to the 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. Unlike Hercule, the performance bugs Paradyn targets are not in direct relation to parallel program design and not intended for explanation of high-level causes. In [10], a cause-effect analysis approach is proposed to explain inefficiencies in distributed programs. It interprets performance losses by comparing earlier execution paths of behaviorally inconsistent processes. Similarly, [17] looks for cause of communication inefficiencies in message passing programs by classification. They train decision trees with real trace data in order to classify individual communication operations and find inefficient behaviors automatically.

Several research work use parallel computational models in performance modeling and evaluation. [5] evaluates the performance of parallel programs coded in algorithmic skeletons with process algebras. POETRIES [7] 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 models based on the structures and attributes performance degradation to parameters composing the models. Hercule differs to POETRIES in that, first, it targets performance explanation and, second, it features a knowledge-based inference system that diagnoses performance in an automated manner.

The project closest to Hercule is Kappa-Pi [8]. This is an automatic performance analysis tool that encodes knowledge about commonly-seen performance problems into deduction rules at various abstraction levels. It explains the problem found by building an expression of the highest-level deduced fact which includes the situation found, the importance of such a problem, and the program elements involved in the problem. Kappa-Pi has been applied to the Master-Worker problem with excellent success. Our work builds on the Kappa-Pi objectives by proposing a systematic approach to extracting knowledge from high-level parallel design patterns.

6 Conclusions and Future directions

This paper describes 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 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.

The Hercule framework offers a prototype performance diagnosis system based on computational patterns. We demonstrated the use of Hercule on the Master-Worker pattern to validate the approach. However, there is still much

work to be done for further improvement and application of model-based diagnosis. First, we are encoding additional parallel patterns, such as *Wavefront* and *Divide-and-Conquer*. As parallel applications can use a combination of parallel paradigms, an important target for our future work is the inclusion of compositional patterns that allow hierarchical reasoning about performance problems.

References

1. Clips. `http://www.ghg.net/clips/CLIPS.html`.
2. Tau tuning and analysis utilities. `http://www.cs.uoregon.edu/research/paracomp/tau/tautools/`.
3. Vampir. `http://www.pallas.com/e/products/index.htm`.
4. P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. on Computer Systems*, 13(1):1–31, 1995.
5. A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Evaluating the performance of skeleton-based high level parallel programs. In *M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, The International Conference on Computational Science (ICCS 2004), Part III, LNCS*, pages 299–306. Springer Verlag, 2004.
6. N. Carriero and D. Gelernter. How to write parallel programs: A guide to perplexed. *ACM Computing Surveys*, 21(3):323–357, 1989.
7. E. Csar, J. G. Mesa, J. Sorribes, and E. Luque. Modeling master-worker applications in poetries. In *HIPS 2004*.
8. A. Espinosa. *Automatic Performance Analysis of Parallel Programs*. PhD thesis, Computer Science Department, University Autonoma de Barcelona, Spain, 2000.
9. T. Fahringer and C. Seragiotto. Automatic search for performance problems in parallel and distributed programs by using multi-experiment analysis. In *International Conference On High Performance Computing (HiPC 2002), Bangalore, India, December 2002. Springer Verlag*.
10. W. M. Jr., T. Leblanc, and V. Almeida. Using cause-effect analysis to understand the performance of distributed programs. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, 1998.
11. L. Li and A. Malony. Knowledge engineering for automatic parallel performance diagnosis. submitted to *Concurrency and Computation: Practice and Experience*.
12. A. Malony and B. Helm. A theory and architecture for automating performance diagnosis. *Future Generation Computer Systems*, 18, 2001.
13. B. Massingill, T. Mattson, and B. Sanders. Patterns for parallel application programs. In *6th Pattern Languages of Programs Workshop*, 1999.
14. B. Massingill, T. Mattson, and B. Sanders. Some algorithm structure and support patterns for parallel application programs. In *9th Pattern Languages of Programs Workshop*, 2002.
15. B. Miller and et al. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.
16. F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Domputing*. Springer-Verlag, 2003.
17. J. Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In ACM International Conference on Supercomputing 2002.
18. J. Yan. Performance tuning with AIMS — an Automated Instrumentation and Monitoring System for multicomputers. In *Proc. 27th Hawaii International Conference on System Sciences*, pages 625–633, 1994.