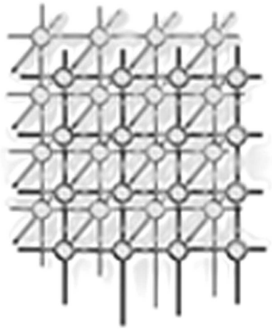
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **00**:1–7



Knowledge Engineering for Automatic Parallel Performance Diagnosis

L. Li*,† and A. D. Malony

Department of Computer and Information Science, University of Oregon, U.S.A

SUMMARY

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. We present a systematic approach to generating performance knowledge for automatically diagnosing parallel programs. Our approach exploits program semantics and parallelism found in computational models to search and explain bugs. We first identify categories of expert knowledge required for performance diagnosis and describe how to extract the knowledge from computational models. Second, we represent the knowledge in such a way that diagnosis can be carried out in an automatic manner. Finally, we demonstrate the effectiveness of our knowledge engineering approach through a case study. Our experience diagnosing Master-Worker programs show that model-based performance knowledge can provide effective guidance for locating and explaining performance bugs at a high level of program abstraction.

KEY WORDS: *Performance; Diagnosis; Knowledge engineering; Parallel*

1. Introduction

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

*Correspondence to: Deschutes Hall Room 228, University of Oregon, Eugene, OR, 97402, USA
†E-mail: lili@cs.uoregon.edu

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 parallelization approach, and the relationship of application parameters to performance. 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*). The goal of automating parallel performance tuning is difficult because the optimization phase can involve expertise that is hard to formalize and automate.

This paper focuses on *parallel performance diagnosis* and how to support it as a automated knowledge-based process. Certainly, important performance measurement and analysis tools, such as Paradyn [16], AIMS [17], and SvPablo [3], have been developed to help programmers diagnose performance problems. Two observations of these tools particularly motivate our work:

1. The performance feedback provided by the tools tend to be descriptive information about parallel program execution at lower levels. 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. Thus, it falls on the users to explain performance observations and reason about causes of performance inefficiencies with respect to computational abstractions used in the program and known only to them. Unfortunately, novice parallel programmers often lack the performance analysis expertise required for high-level problem diagnosis using only lower-level performance data.
2. 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. Typically, the user decides on the instrumentation points and measurement data to collect before an experiment 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 associating effects with causes and managing performance problem investigation ultimately limit diagnosis capability.

We believe that both of the deficiencies above could be addressed by encoding how expert parallel programmers debug performance problems. In particular, we want to capture knowledge about performance problems and how to detect them, and then apply this knowledge in a performance diagnosis system. Where does the performance knowledge come from? The key idea is to extract performance knowledge from parallel computational models that represent structural and communication patterns of a program. The models provide semantically rich descriptions that enable better interpretation and understanding of performance behavior. Our goal is to engineer the performance knowledge in such a way that bottom-up inference of performance causes is supported. In this manner, the diagnosis system can use the performance knowledge base for problem hypothesis testing, whereby a diagnosis search strategy decides which candidate hypothesis is most useful to pursue and new experiment requirements are generated to confirm or deny it.

The answer to whether a performance diagnosis tool would benefit from knowing the computation model of a parallel program is most certainly "yes." The problem we focus on in this paper is the knowledge engineering required for model-based performance diagnosis. The contributions of the performance knowledge derived from parallel models are that they can explain performance loss from high-level program (computation) semantics and provide a sound basis for automating performance diagnosis processes.

The remainder of this paper is organized as follows. The next section presents generic performance diagnosis processes and our model-based diagnosis approach. Section 3 describes how to extract performance knowledge on the basis of computational models. In section 4, knowledge representation method will be discussed. We use Divide-and-Conquer model as an example to demonstrate how to generate performance knowledge in section 5, and present experiment results of diagnosing a Master-Worker test program in section 6. Section 7 provides related research works. The paper concludes with observations and future works in section 8.

2. Model-based Automatic Performance Diagnosis

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. More specifically, the process involves:

- *Designing and running performance experiments.* Researchers in parallel computing have developed integrated measurement systems to facilitate performance analysis [17, 16, 20]. 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 [11]. 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 [12], such as load balance, memory utilization, and communication cost, or tracking down specific source code fragments that are responsible for major performance loss [13]. 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 supporting performance information, synthesize raw performance data to testify the hypotheses, and iteratively refine hypotheses toward higher-level abstractions until some cause is found.

2.2. Model-based Performance Diagnosis Approach

A parallel computational model, also called design pattern [4, 5] or parallel programming paradigm [14] in the literature (we will use these terms interchangeably in this paper), is a recurring algorithmic and communication pattern in parallel computing. Typical models include master-worker, pipeline, divide-and-conquer, and geometric decomposition [5]. The models identify not only computational components and their behaviors (semantics), but process interaction and coordination patterns (parallelism) of runtime behavior of a parallel program. The Divide-and-Conquer (D&C) model, for instance, describes a class of parallel programs that feature recursively splitting a large problem into a certain number of smaller subproblems of the same type until the problem size has been sufficiently reduced, then solving the subproblems in parallel and merging their solutions to achieve the solution to the original problem [4]. *Split*, *solve*, and *merge* are therefore essential computational components in D&C computing. A process interacts with other processes at run time in four different ways – the process receives subproblems from some parent processors if it does not participate in root split, it passes divided subproblems down to some children processes, it receives sub-solutions from children processors to merge, and finally it passes merged solution up to parent processes for further solution join. The interactions dictate parallelism of D&C programs.

Parallel computational models can be supportive of detecting and interpreting performance bugs at a high level of program abstraction. First, *an important aspect of a computational model – communication patterns – reflects data/task dependence relations enforced by parallelization.* In a parallel program, a significant portion of performance inefficiencies (e.g., undesirable communication overheads, load-imbalance, and synchronization cost) is due to process interactions arising from data/task dependency. So the communication pattern can help reveal inherent performance degradation points at parallelism level of the program.

Second, *computational structure and behavioral semantics of a parallel program provide a context for interpreting performance inefficiencies.* To explain a performance degradation point, we need to look up participating processing units and their behaviors causing performance loss at the point. Computation models capture such semantic information.

Third, *model semantics facilitate bottom-up cause inference.* We can classify and synthesize low-level performance data to derive performance metrics of higher-level abstraction, according to behavioral models of a parallel program. Evaluating performance against the metrics drives inferencing process up to a higher abstraction level.

Computational models can also be helpful in performance experiment design and automation of diagnosis process.

- *The structural information helps selectively instrument the program code to capture only relevant performance events.* So model knowledge can effectively constrains the volume of performance information generated for diagnosis, making measurement data more tractable.

- *There is a collection of commonly-used models for constructing real-world parallel applications.* Each of them features a set of typical performance problems and corresponding high-level causes. Researchers have built up rich performance analysis expertise of the models. Performance diagnosis processes involve significant experimentation and reasoning on the basis of prior knowledge. If we can represent and manage the expert performance analysis expertise in a proper way, they will effectively drive diagnosis process with little or no user intervention.

The above potential advantages of computational models motivate our pursuit of *model-based* performance diagnosis approach. The basic idea of the approach is to incorporate program semantics and parallelism embedded in computational models into generic diagnosis processes to address the automation of performance diagnosis at a high level abstraction. We call the information extracted from computational models and required by diagnosis processes *performance knowledge*. The essence of our approach is to use the performance knowledge to automatically drive performance problems search and interpretation. To get there we need to generate performance knowledge from computational models and represent it in an engineered knowledge base.

3. Generation of Model-based Performance knowledge

Experience in knowledge-based system [15] and requirements of performance diagnosis suggest both domain knowledge and inference steps need to be modeled. In this section, we will focus on model-oriented performance knowledge and inference modeling.

3.1. Performance Knowledge

We identify four categories of knowledge provided by computational models.

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 first identify design factors at this abstraction level that are most critical to performance. We investigate factors like task scheduling strategy, data partitioning and mapping methods, model-specific parameters (e.g., the number of workers in Master-Worker model), and other algorithmic factors. A performance cause – an interpretation of performance symptoms in terms of these factors – can therefore immediately direct the programmer to bad parallelism design decisions.

Performance models. Performance models help infer causes from symptoms. Structural information embedded in computational models facilitates performance modeling of parallel programs. In our methodology, a performance model is not a closed-form mathematical formula of system/application parameters. Rather we present descriptive performance compositions that identify computational components/overhead categories and their contribution to overall performance. Given a programming pattern, participating processes can be grouped into clusters in terms of individual process characteristics and their interaction mode with other neighbor clusters. Also distinguishing to traditional performance modeling, we generate a distinct performance model for each process cluster if more than one cluster exist. Performance

models defined in this way serve two goals: (a) performance model of individual process clusters helps detect computational components that account for performance loss. (b) Contrasting performance models of inter-dependent processes to reveal their behavioral differences helps interpret performance losses at process-interacting points such as communication and synchronization (we will illustrate this with Divide-and-Conquer model in section 5.3).

Model-specific performance metrics. Performance diagnosis is driven by metric-based evaluation. 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 [12]. 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 model semantics into their definition. For instance, the metrics could be *process idle time due to waiting for parent processes to split problem* in Divide-and-Conquer model, or *master setup task delay* in Master-Worker model. The advantage of model-specific metrics over generic ones is that they reflect characteristics of problem-solving (i.e. algorithmic properties) and process coordination (i.e. parallelism).

Experiment design and management. Empirical-based performance analysis relies on experiments to capture performance information. The experiments in our method are particularly tailored to performance characteristics as reflected in computational models. Experiment specification includes system parameter setting, instrumentation instruction, and decisions about what performance events to record.

3.2. 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 – are performed in an iterative manner. Figure 1 represents our iterative diagnosis methodology.

The inference process begins with defining model-specific performance metrics that have close affinity with 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 model-specific metric against the expected value or the tolerance for its severity. If the symptom can be directly interpreted by some 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 search space for further search. New experiment specifications and performance metric choices are generated as a result of refinement. They are fed into next iteration of inference.

Refining 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. The process implicitly involves generating

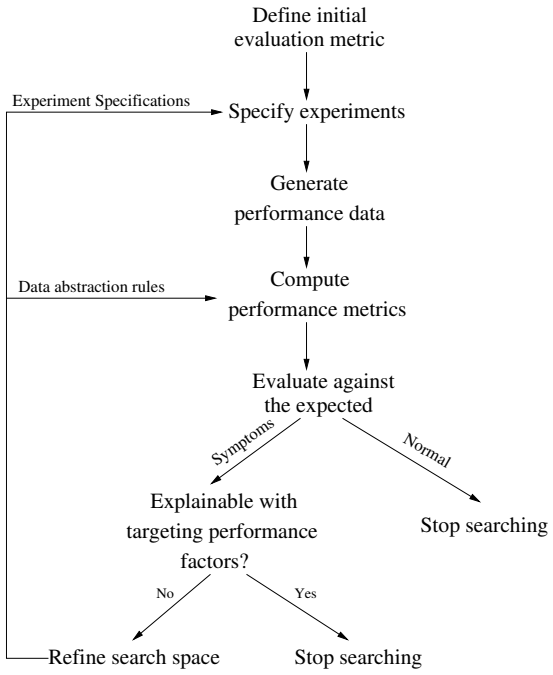


Figure 1. Iterative model-based diagnosis processes.

performance knowledge (models and metrics) of more explanation power. Approaches to performance modeling and metric formulating therefore play a critical role in the refinement of search space. We identify following approaches:

(M1) *breadth decomposition* – decomposing performance cost according to computational components of the computational model;

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

(M3) *concurrency coupling* – formulating performance coupling among interactive processes which arises from concurrency, workload distribution, data/task dependency;

(M4) *parallelism overhead formulating* – identifying and formulating parallelism-specific performance overhead, such as idle time due to task scheduling, workload migration.

We will illustrate the applications of these approaches in the case study section. In addition to above approaches, performance metrics can also be tailored to characteristics of a computational model. For example, the metric *levels of problem split a process being*

delayed, H_{B_f}, in section 5.4 is specific to Divide-and-Conquer model. While it is not a value of performance cost, it helps interpret source of the cost. The metrics are computed by synthesizing raw performance measurement data in terms of their definition rules.

4. Performance Knowledge Representation

Given a parallel computational model, and a symptom associated with a model-specific performance metric of concern, we can create an *inference tree* that represents our bottom-up performance diagnosis approach. An example inference tree of a Divide-and-Conquer model is presented in Figure 3. The root of the tree represents the symptom that we are going to diagnose, branch nodes represent *intermediate observations* (i.e., a performance evaluation with respect to some performance metric, such as waiting time is a *significant* percentage of total elapsed time) that we have achieved so far and need further performance evidences to explain, and leaf nodes represent an explanation of the root symptom in terms of high-level performance factors. Inference processes presented in the tree are driven by metric evaluation. Performance knowledge at various abstraction levels are recalled only when necessary at appropriate node levels. Inference trees, therefore, formalize a structured knowledge invocation process. In addition, inference trees can readily incorporate knowledge generated from new experience or further performance model refinement through adding branches at appropriate tree levels, making knowledge representation highly extensible.

We then encode the inference tree with production rules. 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. In performance diagnosis terms, a rule consists of one or more performance assertions, and performance evidences that must be satisfied to prove the assertions. We make use of syntax defined in the CLIPS [18] expert system building tool to describe production rules. The syntax has 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. 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 reasoning.

Inference trees already encode the content sketch and control structure for knowledge reasoning, so we can immediately transform information embedded in the trees into production rules. For the purpose of automatic performance diagnosis with minimum user intervention,

we need additional rules supporting the main reasoning thread presented in an inference tree. The supporting rules fall into four categories – process clustering, abstract event recognition, performance metric computing, and experiment specification.

For the purpose of semantic-oriented metric computing, participating processes in a computational model should be grouped into process clusters based on the behavioral characteristics of individual processes and their interaction modes with other neighbor clusters. Process clustering rules identify members of each behaviorally distinct process cluster (e.g., in Master-Worker model, which processes play the role of master and which of worker) or inter-dependent process clusters with respect to a particular process (e.g., in divide-and-conquer case, who are parent processes that pass divided sub-problems down to a process, and who are child processes that pass sub-solutions up to the process for merge).

An abstract event consists of a set of related low-level primitive performance events, which together represent a higher-level abstract behavioral pattern, and performance attributes derived from interactions among the events. Abstract events help identify the performance inefficiencies distinct to behavior patterns a computational model exhibits at runtime, therefore facilitating the computing of model-specific performance metrics. Productions rules for recognizing abstract events scan an event trace, generate abstract event instances, and calculate and assign performance attribute values to the instances. The third category – performance metric computing rules – defines how performance attributes of the abstract event instances are synthesized to produce model-specific metrics that can be used to promote performance cause inference. The last category – experiment specification rules – is devoted to the generation of performance data required for the performance metrics. Due to the limitation of space, we refer readers to [26] for details of the representation and examples of the supporting rule categories.

5. Case study – Divide-and-Conquer model knowledge generation

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

5.1. Model description

A wide range of important problems have efficient solution based on the Divide-and-Conquer method. The traditional D&C approach consists of: (i) recursively splitting the problem into a certain number of smaller subproblems of the same type until the problem size has been sufficiently reduced, (ii) solving the subproblems independently, and (iii) merging their solutions to achieve the solution to the original problem [2, 4]. A D&C computation can therefore be viewed as a process of expanding and collapsing a D&C tree [1]. The root of the tree represents the whole problem. 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. In general, D&C algorithms are parallelized by solving the subproblems concurrently at the upper levels of recursion. Figure 2 illustrates a D&C tree which is concurrently computed by four processors. Doing efficient parallel D&C

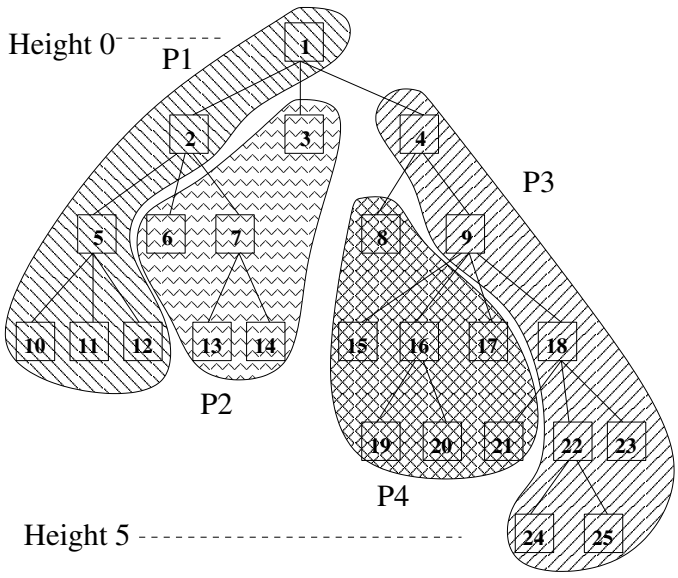


Figure 2. Distributing a D&C tree over 4 processors, where each processor takes care of both splitting, computing, and merging associated with the assigned tree nodes.

computing is nontrivial. During the tree expansion, some nodes can be split into a number of subproblems, while other nodes at the same or higher recursion level stop splitting due to property of the data that have been computed so far, making the D&C tree unbalanced or irregular. Usually, certain task scheduling/migration strategy is used to balance work load among processors. For diagnosing this class of applications, first we need to identify factors and design decisions critical to parallel D&C performance.

5.2. Performance factors

The major issue concerning parallel D&C performance is to balance computation among processors while minimizing communication costs. In general, following factors have most impact on D&C performance:

- *Amount of work at split and merge stage.* If split or merge operations are pretty expensive, then the processors that are assigned nodes at the lower levels of recursion will be idle for a significant amount of time while the nodes at the upper levels are splitting or merging, therefore insufficiently utilized. In this case, dividing a problem into a number of subproblems is often more efficient than that of a few subproblems since the former introduces higher degree of concurrency. The factor therefore has close relevance to the optimal number of children of a branch node in a D&C tree.

- *Size of base problem instance.* The factor decides the height of D&C tree. Recall that 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 depth of recursive split and amount of base problems.
- *Process granularity.* As mentioned above, we should seek a balancing point between the number of processors used for parallel computing and other solution-specific parameters (e.g. granularity of split and height of expansion tree) induced from D&C to achieve satisfactory performance.
- *Scheduling algorithms used for balancing workload.* The factor decides the degree of load balance and communication cost of moving computation around. Without loss of generality, and to simplify later discussion, we make two assumptions about the scheduling algorithms: 1. load migration occurs only when there are idle processors available for doing extra computation. That is, at any instance of time, if all processors are busy then no workload migration can happen. 2. If a processor passes some of its subproblems (child branches in D&C tree) to other processors, then solutions to the subproblems will be sent back to the processor to merge as soon as they are computed. The assumptions facilitate localized modeling of performance loss due to scheduling inefficiency in the next section.

5.3. Performance models

Next we model D&C performance by referring to the rules defined in section 3.2. According to breadth decomposition rule, total elapsed time of a single processor p in a parallel D&C execution, denoted as t_p, consists of t_{setup} (process-setup overhead), t_{split} (amount of time spent splitting problems), t_{solve} (amount of time time spent solving base problem cases), t_{merge} (amount of time spent merging sub-solutions), t_{comm} (communication time for transferring data among processors), t_{wait} (amount of time spent waiting for workload assignment or synchronizing with other processors), and $t_{housekeeping}$ (amount of software housekeeping time with respect to task scheduling or load balancing algorithm):

$$(\mathbf{M1}) \Rightarrow t_p \quad = \quad t_{setup} + t_{split} + t_{solve} + t_{merge} + t_{comm} + t_{wait} + t_{housekeeping} \qquad (1)$$

Whenever we refer to communication time in the paper, 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, hence parallelism design.

Performance coupling of processors in D&C computing manifests in four aspects. First, the processors which do not participate in root split need to wait for subproblems to be passed down from some parent processor (P_{parent}). Second, the processors which are idle at some point will get workload transferred from some neighbor processor ($P_{neighbor}$) that is designated by the task scheduling algorithm. Third, the processors need to get sub-solutions from children processors (P_{child}) to merge. And last, after a processor finishes merging assigned local nodes,

it needs to wait the parent processors dealing with upper level nodes to finish their merge operation before finalizing. The four types of interdependency therefore dictate main sources of performance loss.

$$(\mathbf{M3}) \Rightarrow t_{wait} \quad = \quad t_{w-split} + t_{w-migration} + t_{w-merge} + t_{w-finish} + t_{others} \tag{2}$$

In above equation, the first four items correspond to the waiting time with respect to the four types of process interaction. The last item accounts for performance loss that cannot be mathematically represented. The loss is primarily due to local load-imbalance. For instance, if the branches assigned to $p3$ in Figure 2 involve more workload than that of $p2$, the load-imbalance will give rise to extra idle time of $p2$ before parent processor $p1$ gathers all sub-solutions and performs merge at node 1, which accounts for t_{others} of $p2$.

Likewise, communication overheads t_{comm} – the amount of time spent transferring data, N_{comm} – the number of times of interprocessor communication, and V_{comm} – the volume of communication data can be decomposed into three categories according to computation phases:

$$(\mathbf{M2}) \Rightarrow t_{comm} = t_{comm-split} + t_{comm-migration} + t_{comm-merge} \tag{3}$$

$$(\mathbf{M2}) \Rightarrow N_{comm} = N_{comm-split} + N_{comm-migration} + N_{comm-merge} \tag{4}$$

$$(\mathbf{M2}) \Rightarrow V_{comm} = V_{comm-split} + V_{comm-migration} + V_{comm-merge} \tag{5}$$

5.4. Performance metrics

The performance models above enable the definition of model-specific metrics to use for evaluation. We start with evaluating process efficiencies to detect a top-level symptom. In terms of D&C semantics, process efficiency is defined as:

$$\textbf{efficiency} := \frac{(t_{split} + t_{merge} + t_{solve})}{t_p} \tag{6}$$

Let B_f and B_l be the first and the last tree branch processor p deals with during an execution, then we define $\boldsymbol{H_{B_f}}$ – *levels of problem split process p being delayed* – as the height of B_f root and $\boldsymbol{H_{B_l}}$ – *levels of solution merge process p being delayed* – the height of B_l root. We call processor p' a parent of p if p' deals with higher level expansion tree nodes that lie on the path from tree root to root of branch B_f. Let $\{p^1_{parent}, \ldots, p^K_{parent}\}$ be parent processor set of p, it implies that p^1_{parent} splits tree root and passes sub-problems to p^2_{parent}, p^2_{parent} to p^3_{parent}, ..., and finally p^K_{parent} passes B_f to p. For example, parent set of $p4$ in Figure 2 is $\{p1, p3\}$. We call processor p' a neighbor of p if p' migrates some branch to p to balance workload. Let $\{p^1_{neighbor}, \ldots, p^M_{neighbor}\}$ be the neighbor processor set of p, and $p^M_{neighbor}$ the processor that migrates branch B_l to p. Neighbor set of $p4$ in Figure 2, for instance, is $\{p3\}$. We call processor p' a child of p if p passes/migrates some subproblem branch to p'. Let $\{p^1_{child}, \ldots, p^N_{child}\}$ be the children processor set of p. Children set of $p1$ in Figure 2, for instance, is $\{p2, p3\}$. Referring to the refining rules in section 3.2, we formulate metrics with respect to each item in equation

(2) as below[†]:

$$(\mathbf{M2}, \mathbf{M3}) \Rightarrow t_{w-split} := \sum_{i=0}^{K} t^i_{split} \tag{7}$$

where t^i_{split} represents the amount of time processor p^i_{parent} spent splitting nodes on the path from the root down to branch B_f;

$$(\mathbf{M3}, \mathbf{M4}) \Rightarrow t_{w-migration} := \sum_{i=1}^{M} t^i_{w-migration} \tag{8}$$

where $t^i_{w-migration}$ is the amount of time waiting processor $p^i_{neighbor}$ to transfer load;

$$(\mathbf{M2}, \mathbf{M3}) \Rightarrow t_{w-merge} := \sum_{i=1}^{N} t^i_{w-merge} \tag{9}$$

where $t^i_{w-merge}$ is the amount of time spent waiting for processor p^i_{child} to pass sub-solution; Let $\overline{p}^1_{parent}, \ldots, \overline{p}^{K'}_{parent}$ be parent processors of processor $p^M_{neighbor}$. Recall that $p^M_{neighbor}$ passes the branch B_l to p, then

$$(\mathbf{M2}, \mathbf{M3}) \Rightarrow t_{w-finish} := \sum_{i=1}^{K'} t^i_{merge} \tag{10}$$

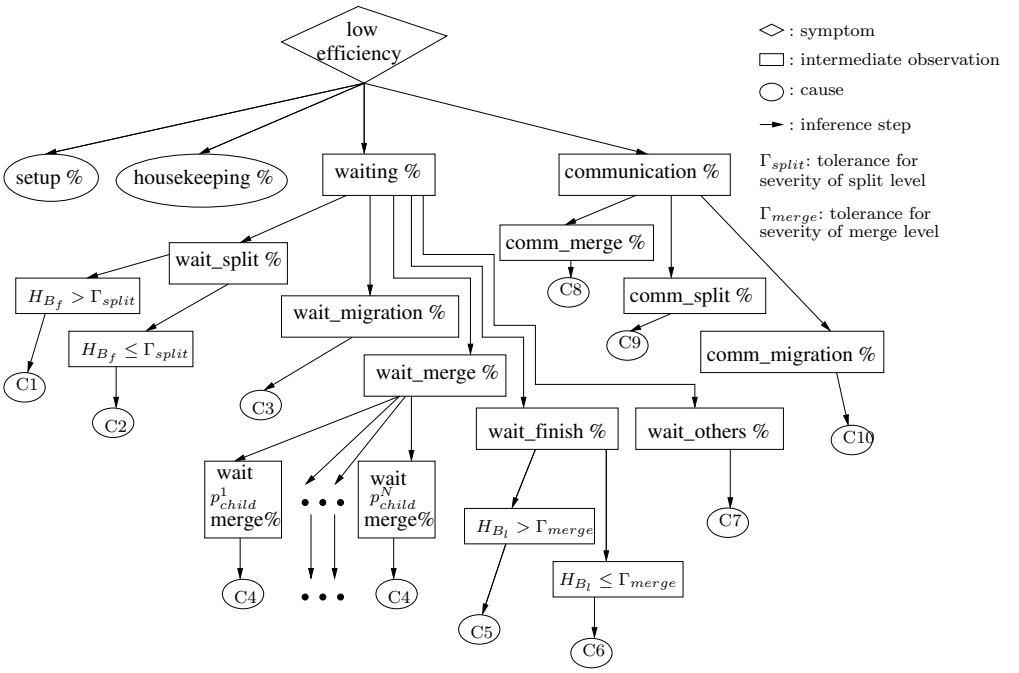
where t^i_{merge} is the amount of time processor $\overline{p}^i_{parent}$ spend merging nodes on the path from branch B_l up to the root. Illustrating the metrics with processor $p2$ in Figure 2, we have:
$H_{B_f} = 1$,
$H_{B_l} = 2$,
$t_{w-split}$ = node 1 split,
$t_{w-migration} = t^{p1}_{w-migration}$, i.e., waiting time for $p1$ to migrate node 6 and node 7,
$t_{w-merge} = 0$, since $p2$ does not have children processors,
$t_{w-finish}$ = node 2 merge + node 1 merge,

Decompositional performance modeling and metric computing in terms of model semantics greatly facilitates generation of diagnosis knowledge.

5.5. Generation of model knowledge

An inference tree of diagnosing D&C programs is presented in Figure 3, referring to above performance models and metrics. We can see that the model-specific metrics effectively help guide the reasoning from low-level symptoms to causes at high level of abstraction. For example, ratio of waiting time due to parent node split to overall performance loss

[†]Model implementation variation may enforce other minor overheads in following waiting time categories.



c1: The process was idle until H_{B_f} levels of split. There might be more processors used than necessary for optimal performance.
c2: Split is pretty expensive. Try to split problem into more subproblems to increase concurrency.
c3: The process can be better utilized if the scheduling algorithm can reduce load migration time.
c4: There is local load imblanace between p and p^i_{child}. The branch transferred to p^i_{child} incurred more work load than the counterpart in p.
c5: The process was idle during upper H_{B_l} levels of merge. There might be more processors used than necessary for optimal performance.
c6: Merge is pretty expensive. Try to split problem into more subproblems to increase concurrency.
c7: The performance loss here is due to load imbalance between process p with the other processors than p_{parent}, $p_{neighbor}$, and p_{child}.
c8: Communication cost for merge is expensive. $N_{comm-merge}$ and $V_{comm-merge}$ help explain the cost.
c9: Communication cost for split is expensive. $N_{comm-split}$ and $V_{comm-split}$ help explain the cost.
c10: Communication cost for migration is expensive. $N_{comm-migration}$ and $V_{comm-migration}$ help explain the cost.

Figure 3. Inference Tree for Performance Diagnosis of Divide-and-Conquer model.

(i.e., wait_split% in Figure 3) and the levels of split being delayed (H_{B_f}) jointly distinguish the performance impact of *split* operations. As inference going deep, causes of performance inefficiency are localized. Waiting time for merge (wait_merge), for instance, is broken down into waiting time for individual child processors, which helps isolate local load imbalance between processor p and specific child processors.

The knowledge inference present in Figure 3 is not meant to be complete. There is still room for further refining search for performance bugs. For instance, since processor p may

pass branches of workload down to a child processor p^i_{child} for multiple times, $t^i_{w-merge}$ – the amount of time spent waiting processor p^i_{child} to send back sub-solutions – can be further decomposed into waiting time with respect to each pass-down, so that we are able to attribute local load-imbalance to specific branches of involved processes. Nevertheless, designed to be extensible, our inference tree 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 (we will illustrate the experiment variance in section 6.3). Our diagnosis system can construct the experiments accordingly to collect performance data of the related metrics.

6. Experimentation

6.1. Performance Diagnosis and Validation System

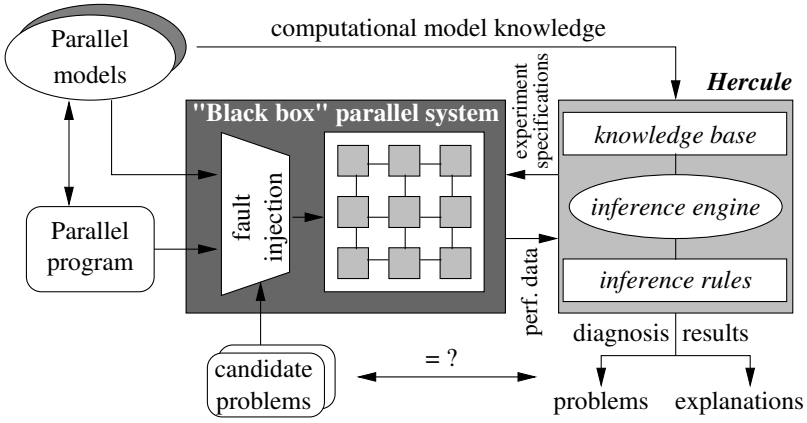


Figure 4. Hercule and performance diagnosis validation system.

We have built an automatic performance diagnosis (PD) system called *Hercule* based on the knowledge engineering and performance problem inferencing approach discussed in previous sections [‡]. In this section, we demonstrate Hercule's ability to diagnosis performance problems in a Master-Worker parallel program run on an distributed memory cluster with 16 dual-processor nodes and a gigabit ethernet switch.

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

[‡]We chose the name Hercule in keeping with the spirit of our earlier performance diagnosis project, *Poirot* [24].

desired forms. However, in addition, a validation environment should also support the injection of known (model level) performance problems in a parallel program.[§] 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 shows how Hercule and the validation environment we are developing work together, as new parallel computation models are included. The core of automated performance diagnosis is a knowledge base that stores encoded, retrievable performance knowledge of computational models. Given a program to be diagnosed, Hercule starts with being informed of the computational model the program is patterned on, then comes up with a set of experiments as it explores performance hypotheses. 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. We then compare the conclusions against the performance problems introduced at the start.

6.2. Performance Problem Test Program

Master-Worker (M-W), also referred to as the *Master-Slave* [5], is one of the most often used parallel computational models. In this model a computation 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 challenges in M-W computational model are to organize the task assignment so that all workers finish the last task processing at approximately the same time (*time balancing*), and the total elapsed time of execution is minimized (*makespan minimization*) [7]. The master generally employs certain task scheduling algorithms to achieve these goals [8]. We can express these as performance goals to achieve.

With each task to be computed on a worker processor, we associate three activities [¶]:

1. *Worker request.* A worker sends a requesting message to the master upon finishing the last task processing, encapsulating the last task results in the message. Note that the worker does not need to request its first task.
2. *Task setup.* Upon receiving a worker request, the master runs the task scheduling algorithm to decide a task assignment, collects the data needed by the task, and transfers the data and task specification to the requesting worker. If the task setup cost is expensive relative to average task processing time, or there are too many workers to be effectively served by the master, it will result in the master being seen, in some time periods, as a performance bottleneck where a number of workers get stuck waiting for task assignments.

[§] *Fault injection* is a common part of software testing and diagnosis environments.
[¶] There are different types of worker behavior in other M-W variants, such as in [6].

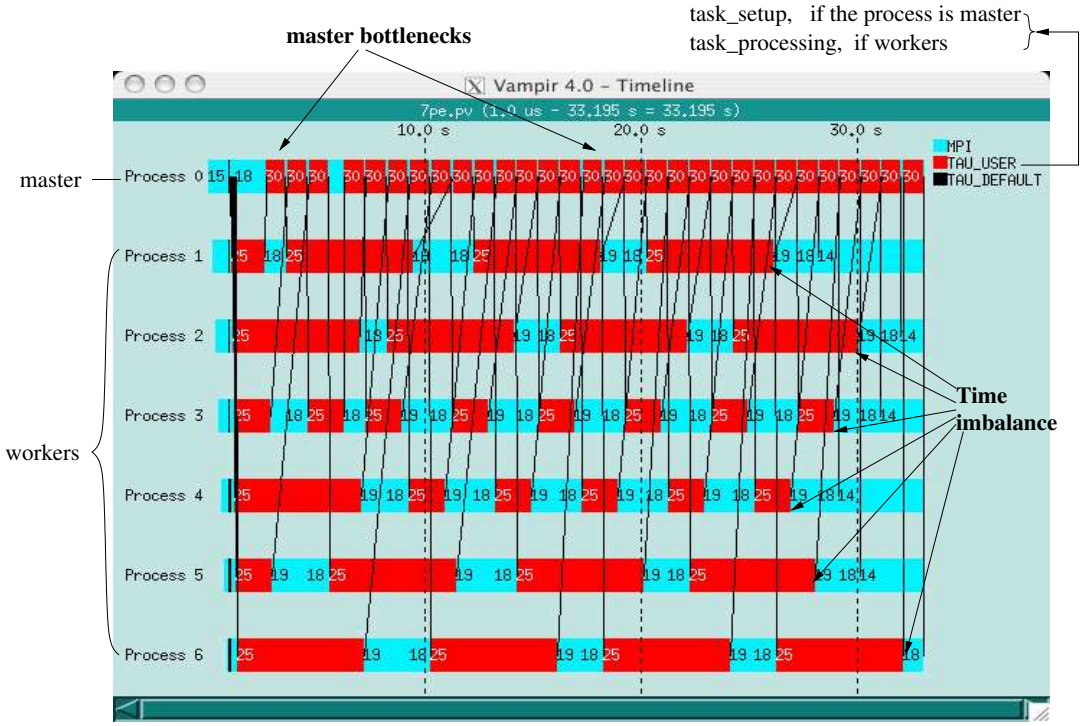


Figure 5. Vampir timeline view of an example M-W program execution.

3. *Worker computing*. The worker receives the task information and starts the assigned computation.

Due to the limitation of space, we refer readers to [26] for details of diagnosis knowledge generation of the Master-Worker model.

We created a synthetic parallel Master-Worker program to demonstrate our knowledge-based performance diagnosis approach. The performance problems we introduce in the program focus on the impact of master-setup-task 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 small 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 and 6 respectively present a Vampir [28] timeline view and ParaProf [29] profile display of an execution of the program with 7 processors. The event trace and profiles are generated by the TAU [20] performance measurement system with only major model components being instrumented. In Figure 5, 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

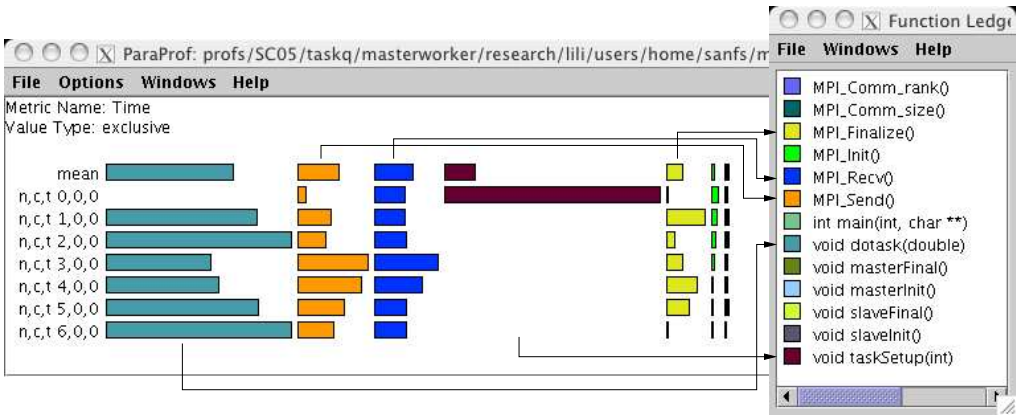


Figure 6. Graphical display of relative time spent in each function on each node, context, thread.

Table I. Performance metrics of the test Master-worker program.

Metric names	Performance loss%
waiting time in master bottlenecks	39.2%
waiting time for the master to set up tasks	34.3%
waiting time for slow workers to finish computing before finalization	14.8%
effective communication time	6.2%

figures, blocking/waiting time of processors is implicitly included in elapsed time of blocked **MPI_Send**, **MPI_Recv** and **MPI_Finalize** operations.

6.3. Diagnosis Results

Given the program and performance knowledge associated with M-W model, Hercule will automatically request three experiments during the diagnosis. The first experiment collects data for computing efficiencies of each worker (i.e., task processing time/total elapsed execution time). 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. The second experiment generates performance information for identifying overhead categories with respect to computation components of the M-W model and their contributions. Waiting (idle) time stands out as a result of this iteration of inference. The third experiment then targets model-specific metrics in Table (I). The metrics are presented in the form of percentage that each performance loss category contributes to the overall performance loss (i.e. total elapsed execution time minus effective task processing time). It is interesting to note that waiting time in master bottlenecks accounts for 39.2% lost cycles. To identify whether the amount of workers or task setup speed is the dominant factor causing master bottlenecks, Hercule tests


```
dyna6-166:~/PerfDiagnosis/classes 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 waiting (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 its computation (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
the 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.
--------------------------------------------------------------------------
```

Figure 7. Diagnosis result output of the example M-W test program.

another model-specific metric – *severity of master bottleneck* – which is the average number of worker requests got stuck in master bottlenecks. The metric turns out to be 1.6, which will not cause severe delay unless a single task setup time is expensive relative to task processing time. Given that the waiting time for task setup accounts for 34.3% lost cycles, it is more likely that the low master capacity of setting up tasks is the main reason causing master bottlenecks. The inference process and diagnosis results are presented in Figure 7, in a manner close to programmer's reasoning and understanding.

7. Related work

Paradyn [16] 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. 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 [21] is an implementation of the *Performance Specification Language* (PSL) developed by the APART project [22]. Using syntax and semantics of Java programming language, JavaPSL is intended for flexibly defining performance properties. A bottleneck analysis tool using JavaPSL can automatically search for bottlenecks by navigating through the performance data space and computing pre-defined performance properties. In distinct to JavaPSL, our performance metrics are defined in terms of model-semantics, and intended for both automatic performance bug search and interpretation.

Poirot [24] is an adaptable and automated performance diagnosis architecture. Instead of performance knowledge engineering, it focuses on gathering a variety of performance diagnosis methods and selecting method for adaptable diagnosis.

In [25], a Cause-Effect analysis approach is proposed to explain inefficiencies in distributed programs. The approach detects local performance losses occurring in some execution periods, which are often a result of behavioral inconsistence between two or more processors. It interprets the performance losses by comparing earlier execution paths of the inconsistent processors.

In [27], an approach to looking for the cause of communication inefficiencies in message passing programs is presented. In the approach, they train decision trees with real performance tracing data in order to automatically classify individual communication operations and find inefficient behaviors.

Kappa-Pi [23] is also a rule-based automatic performance analysis tool. In this tool, knowledge about commonly-seen performance problems is encoded into deduction rules at various abstraction levels. It explains the problem found to the user by building an expression of the highest level deduced facts which includes the situation found, the importance of such a problem, and the program elements involved in the problem. 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.

8. Conclusions and Future directions

This paper describe 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. Four major categories of knowledge are identified: performance models, model-specific metrics, high-level performance factors, and experiment designs. Our method addresses how to extract expert information in each category for particular parallel models of interest. The methodology also advocates model-based inference by refining performance models and metrics. One important objective of our work is to study how diagnosis knowledge is represented. In this work, we showed the use of CLIPS for knowledge engineering. We also demonstrate knowledge generation of Divide-and-

Conquer model and the use of the prototype Hercule parallel performance diagnosis system on a representative programing paradigm, Master-Worker. Our preliminary results show that model-based performance knowledge provides effective guidance for locating and explaining performance bugs at a high level of program abstraction.

We have built knowledge bases of Master-Worker and Divide-and-Conquer model with the methods presented in this paper. There is still much work to be done for further improvement and application of this approach. First, we will extend our inference trees with diagnoses of performance problems with respect to inter-play of a computational model with the underlying parallel machine. This will require a greater degree of definition in the performance metrics and analysis. Second, we will create knowledge bases for additional parallel models, such as *Fork-Join*, *Phase-based*, and possibly *BSP*. Parallel applications can use a combination of parallel paradigms. An important target for our future work is the inclusion of compositional modeling. This will add another level of complexity to the knowledge engineering and problem inferencing since we must be able to reason about the interplay of one model with another. Compositional models will naturally include a hierarchical modeling requirement. Finally, we will continue to enhance the Hercule system and build a more powerful validation system. The latter will be important in testing the power of the model-engineered diagnosis knowledge developed in the Hercule environment.

ACKNOWLEDGEMENTS

We would like to thank Professor Ginnie Lo for reviewing and valuable comments on earlier versions of this paper.

REFERENCES

1. I-Chen Wu, H. T. Kung. Communication complexity for parallel divide-and-conquer. In *Proceedings of the 32nd annual symposium on Foundations of computer science*, 1991
2. Robert Sedgewick. *Algorithms*. Addison Wesley, 2nd edition, 1988
3. SvPablo, University of Illinois, `http://www.renci.unc.edu/Project/SVPablo/SvPabloOverview.htm` [April 10 2005]
4. B. L. Massingill and T. G. Mattson and B. A. Sanders. Some Algorithm Structure and Support Patterns for Parallel Application Programs. In *Proc. 9th Pattern Languages of Programs Workshop*, 2002.
5. B. L. Massingill and T. G. Mattson and B. A. Sanders. Patterns for Parallel Application Programs. In *Proc. 6th Pattern Languages of Programs Workshop*, 1999
6. E. Cesar, J.G. Mesa, J. Sorribes, and E.Luque. Modeling Master-Worker Applications in POETRIES. In *Proc. of the 9th International Workeshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.
7. Gary Shao, Fran Berman, Rich Wolski. Performance Effects of Scheduling Strategies for Master/slave Distributed Applications. *International Conference on Parallel and Distributed Processing Techniques and Applications*, June, 1999.
8. Sartaj Sahni. Scheduling Master-Slave Multiprocessor Systems. *IEEE transactions on Computers* 1996; 45(10):1195-1199.
9. R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics* 1969; 17(2):416-429.
10. P. Brucker, C. Dhaenens-Flipo, S. Knust, S.A. Kravchenko, F. Werner. Complexity results for parallel machine problems with a single server. *Journal of Scheduling* 2002; 5:429-457.

11. A. Grama, A. Gupta, G. Karypis and V. Kumar. Analytical Modeling of Parallel Programs. In *Introduction to Parallel Computing*. Addison-Wesley, 2003,
12. T. Fahringer and C. S. Jr. Aksum: a tool for multi-experiment automated searching for bottlenecks in parallel and distributed programs. In *Proceedings of SC2002*, 2002.
13. John Mellor-Crummey and Robert Fowler and Gabriel Marin and Nathan Tallent. HPCView: A Tool for Top-down Analysis of Node Performance. *The Journal of Supercomputing* 2002; 23:81-104.
14. N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to Perplexed. *ACM Computing Surveys* 1989 21(3):323-357.
15. D. A. Waterman, A Guide to Expert Systems. *Reading, MA*, Addison-Wesley, 1985.
16. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, T. Newhall. The Paradyn Parallel Performance Measurement Tool. In *IEEE Computer* 1995; 28(11):37-46.
17. J. C. Yan. Performance tuning with AIMS — an Automated Instrumentation and Monitoring System for multicomputers. In *Proc. 27th Hawaii International Conference on System Sciences*, pp.625–633, 1994.
18. CLIPS: A Tool for Building Expert Systems, `http://www.ghg.net/clips/CLIPS.html`. [April 10 2005]
19. P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. on Computer Systems* 1995; 13(1):1-31.
20. TAU - Tuning and Analysis Utilities, University of Oregon, `http://www.cs.uoregon.edu/research/paracomp/tau/tautools/` [April 10 2005]
21. T. Fahringer and C. S. Jnio. Modeling and detecting performance problems for distributed and parallel programs with JavaPSL. In *Proceedings of SC2001*, 2001
22. APART IST Working Group on Automatic Performance Analysis: Real Tools, `http://www.fz-juelich.de/zam/RD/coop/apart/` [April 10 2005]
23. A. Espinosa, Automatic Performance Analysis of Parallel Programs, PhD thesis, Computer Science Department, University Autonoma de Barcelona, Barcelona, Spain, 2000
24. Allen D. Malony and B. Robert Helm, A theory and architecture for automating performance diagnosis. *Future Generation Computer Systems* 2001; 18:189-200.
25. Wagner Meira Jr., Thomas J. Leblanc, and Virglio A. F. Almeida. Using cause-effect analysis to understand the performance of distributed programs. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1998
26. L. Li and A. D. Malony. Knowledge engineering for model-based parallel performance diagnosis. Submitted to Supercomputing 2005.
27. Jeffrey Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *ACM International Conference on Supercomputing 2002.*
28. Vampir, `http://www.pallas.com/e/products/index.htm` [April 10 2005]
29. R. Bell, A. D. Malony, and S. Shende. A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proc. EUROPAR 2003 conference*, LNCS 2790, Springer, Berlin, pp. 17-26, 2003.

