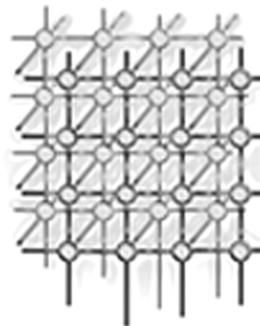
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **00**:1–7



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 parallel programming patterns to search and explain bugs. The approach addresses how to extract the expert knowledge required for performance diagnosis from parallel patterns and represents the knowledge in a manner such that the diagnosis process can be automated. We demonstrate the effectiveness of our knowledge engineering approach through a case study. Our experience diagnosing Divide-and-Conquer programs shows that pattern-based performance knowledge can provide effective guidance for locating and explaining performance bugs at a high level of program abstraction. Copyright © 2005 John Wiley & Sons, Ltd.

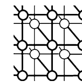
KEY WORDS: *Parallel program; Performance diagnosis; Patterns; Knowledge engineering; Divide-and-Conquer*

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

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

Copyright © 2005 John Wiley & Sons, Ltd.



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 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 *performance diagnosis* and how to support it as an automated knowledge-based process. Certainly, important performance measurement and analysis tools, such as Paradyn [9], AIMS [10], and SvPablo [2], have been developed to help programmers diagnose performance bugs. 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 design patterns that represent structural and communication patterns of a program. The patterns 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 forming problem hypothesis, evaluating performance metrics to test the hypothesis, and deciding which candidate hypothesis is most useful to pursue and new experiment requirements are generated to confirm or deny it.

Copyright © 2005 John Wiley & Sons, Ltd.

The answer to whether a performance diagnosis tool would benefit from knowing the computation and communication pattern of a parallel program is most certainly “yes.” The problem we focus on in this paper is the knowledge engineering required for pattern-based performance diagnosis. The contributions of the performance knowledge derived from parallel patterns 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 pattern-based diagnosis approach. Section 3 describes how to extract performance knowledge on the basis of parallel patterns. We use Divide-and-Conquer pattern as an example to illustrate the generation of performance knowledge in Section §4. Section §5 describes Hercule system that implements the pattern-based performance diagnosis approach. We present Hercule results of diagnosing a Divide-and-Conquer program, parallel Quicksort, in Section §6. Section §7 provides related research works. The paper concludes with observations and future works in Section §8.

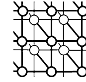
2. Pattern-based Automatic Performance Diagnosis

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 [10, 9, 13]. They observe performance of a parallel program under a specific circumstance with specified input data, problem size, number of processors, and other parameters. The experiments also decide on points of instrumentation and what performance information to capture. Performance data are then collected from experiment runs.
- *Finding symptoms.* We define a *symptom* as an observation that deviates from performance expectation. General metrics for evaluating performance includes execution time, parallel overhead, speedup, efficiency, and cost [5]. By comparing the metrics computed from performance data with what is expected, we can find symptoms such as low scalability, poor efficiency, and so on.
- *Inferring causes from symptoms. Causes* are explanations of observed symptoms. Expert programmers interpret performance symptoms at different levels of abstraction. They may explain symptoms by looking at more specific performance properties [6], such as load balance, memory utilization, and communication cost, or tracking down specific source code fragments that are responsible for major performance loss [7]. Attributing a symptom to culprit causes requires bridging a semantic gap between raw performance data and higher-level parallel program abstraction. Expert parallel programmers, relying

Copyright © 2005 John Wiley & Sons, Ltd.



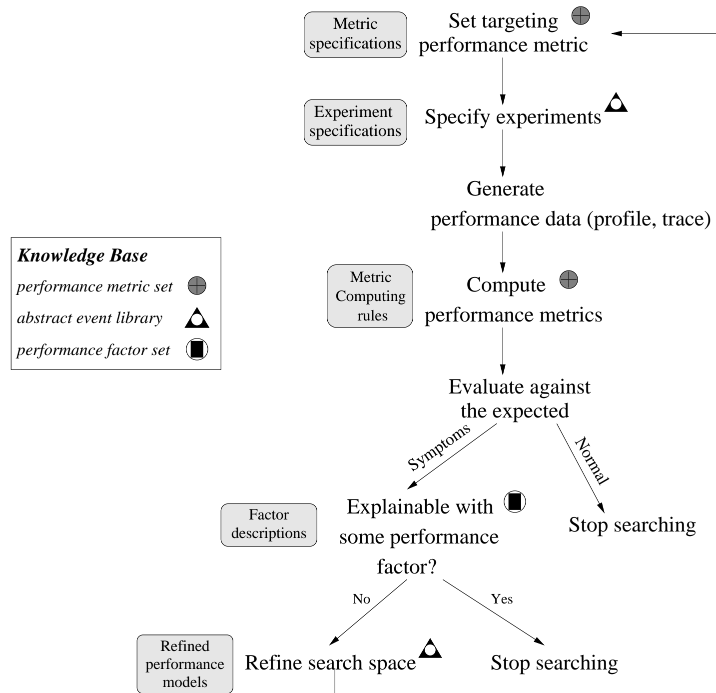


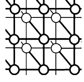
Figure 1: A high-level overview of pattern-based iterative diagnosis processes. The use of performance knowledge derived from patterns is annotated in the process steps.

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 towards higher-level abstractions until some cause is found.

2.2. Pattern-based Performance Diagnosis Approach

A parallel pattern, also called design pattern [3, 4] or parallel programming paradigm [8] in the literature (we will use these terms interchangeably in this paper), is a recurring algorithmic and communication pattern in parallel computing. Typical patterns include master-worker, pipeline, divide-and-conquer, and geometric decomposition [4]. Patterns usually describes computational components of a parallel program and their behaviors (semantics) and how multiple threads of execution interact and collaborate in a parallel solution (parallelism). Parallel patterns 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

Copyright © 2005 John Wiley & Sons, Ltd.



processes tailored to realistic program implementations. Specifically, we envision that patterns 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 an experiment to generate the information. Parallel patterns 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 patterns capture information about computational structures and process coordination patterns generic to a broad range of parallel applications. This information provides a context for describing performance properties and attributing them to associated process behaviors. In the context of pattern-specific behaviors, the low-level performance details can be classified and synthesized to derive performance metrics that have explanation power at a higher level of abstraction.

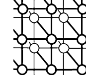
Expert analysis of performance problems. Expert parallel programmers have built up rich expertise in both programming with and analyzing commonly-used parallel patterns. In performance diagnosis, they implicitly refer to their prior knowledge for attributing performance symptoms to causes. If we can represent and manage the expert performance knowledge in a proper manner, they will effectively drive diagnosis process with little or no user assistance.

The above potential advantages of parallel patterns motivate our pursuit of a *pattern-based* performance diagnosis approach. The basic idea of our approach is depicted in Figure 1. Our approach incorporates program semantics and parallelism embedded in parallel patterns into generic iterative diagnosis processes to address the automation of performance diagnosis at a high level abstraction. We call the information extracted from patterns and required by diagnosis processes *performance knowledge*. Performance knowledge about a pattern consists of abstract events that describe pattern behavioral characteristics, pattern-specific metrics to be used to evaluate various performance aspects, and performance-critical design factors associated with the pattern (which we call *performance factors* and form candidate causes for interpreting performance problems). Performance diagnosis processes refer to the knowledge, instead of enforcing intelligent user input, to guide performance cause inference. To get there we need to generate performance knowledge from patterns and represent it in an engineered knowledge base.

3. Performance Knowledge Generation Based on Patterns

Extracting performance knowledge from parallel patterns involves four types of modeling, which are shown in Figure 2. The *behavioral modeling* captures knowledge of program execution semantics as behavioral models represented by a set of abstract events at varying detail levels, depending on the complexity of the pattern and diagnosis needs. The purpose of the abstract

Copyright © 2005 John Wiley & Sons, Ltd.



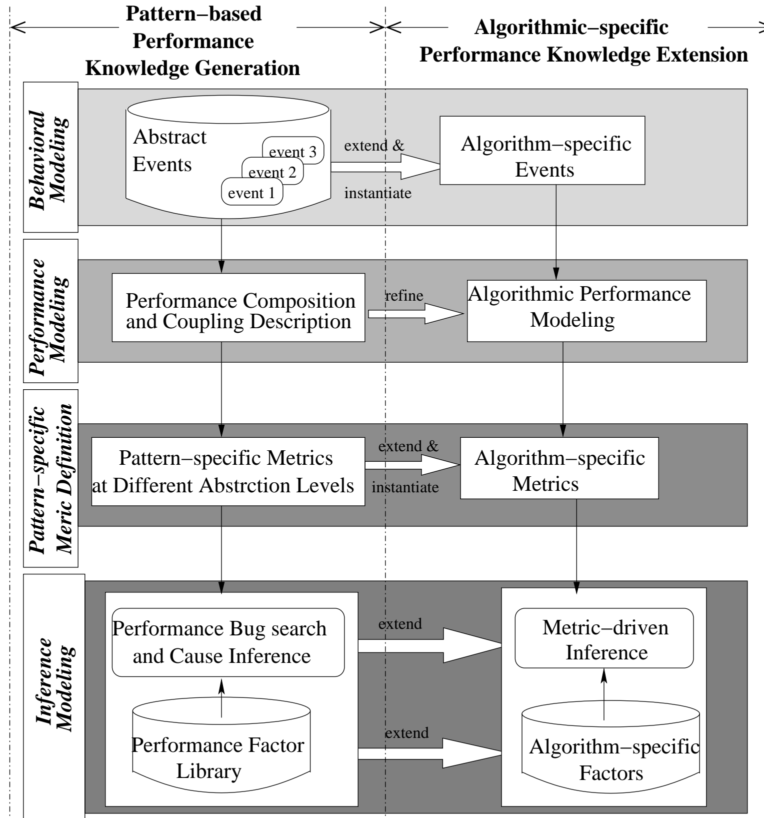
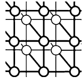


Figure 2: Generating performance knowledge from parallel patterns. Algorithm variants can derive performance knowledge from the basic generic pattern.

events in the diagnosis system is to give contextual informaton for performance modeling, metric definition, and diagnostic inferencing.

We adapt the *behavioral model description* used in EBBA [12] to describe abstract event types. 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., inter-process 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

Copyright © 2005 John Wiley & Sons, Ltd.

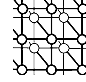


events. For primitive events, the format often takes the form of an ordered tuple that consists of the event identifier, the timestamp when the event occurred, the event location, etc. For constituent abstract events, their types are specified. *Associated events* are a list of related abstract event 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 5 in the next section shows example abstract events for the D&C pattern.

Performance modeling is carried out based on structural information in the abstract events. The modeling identifies performance attributes with respect to the behavior models represented by abstract events and pattern-specific performance overhead categories. *Performance metrics* for evaluating the overhead categories are then defined, in terms of performance attributes in related abstract events. Traditional performance analysis approaches use generic metrics without relevance to program semantics, such as *synchronization overhead* and *imperfect L2 cache behavior* in [6]. A consequence of evaluation with the generic metrics is that the users still need to attribute them to specific program design decisions. To enhance explanation power of performance metrics, we intend to incorporate pattern semantics into their definition. For instance, the metrics could be *communication time due to problem splitting and merging* in Divide-and-Conquer pattern. The advantage of pattern-specific metrics over generic ones is that they reflect characteristics of problem-solving and parallelism.

Inference modeling captures and represents the performance bug search and interpretation process formally. In our diagnosis approach, we aim to find performance causes (i.e., an interpretation of performance symptoms) at the level of the design of the parallel program, that is, performance-critical design factors specific to the particular parallel pattern. The cause inference is therefore the mapping of low-level performance measurement data to high-level performance factors to explain a performance symptom (i.e., an anomaly deviating from the expected). The inference process is captured in the form of an inference tree where the root is the symptom to be diagnosed, the branch nodes are intermediate observations obtained so far and needing further performance evidences to explain, and the leaf nodes are explanations of the root symptom in terms of high-level performance factors. An inference tree for diagnosing symptom "low speedup" in D&C pattern is presented in Figure 6. An intermediate observation is obtained by evaluating a pattern-specific performance metric against the expected value (from performance modeling) or a certain pre-set threshold that defines the tolerance of severity of the performance overhead the metric represents. In Figure 6, for example, node *comm_solve* means the communication cost in the stage of solving problem cases. If it turns out to be significant comparing to the expected, the inference engine will continue to search for the node's child branches. The leaf nodes finally reached together compose an explanation of the root symptom. It is clear that inference processes presented in the trees are driven by metric evaluation. Performance knowledge associated with the metrics, including related abstract events, performance overhead types, and metric computing rules, are recalled only when needed by current inference step. Inference trees, therefore, formalize a structured knowledge invocation process.

Copyright © 2005 John Wiley & Sons, Ltd.



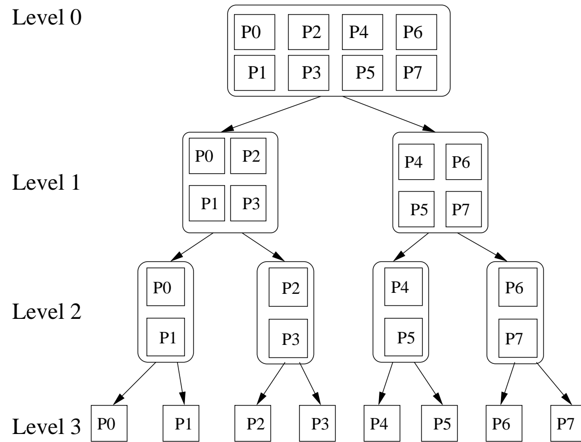


Figure 3: A D&C computation with 8 processors and three levels of problem splitting, where each splitting divides processors into two groups which work on orthogonal data sets independently.

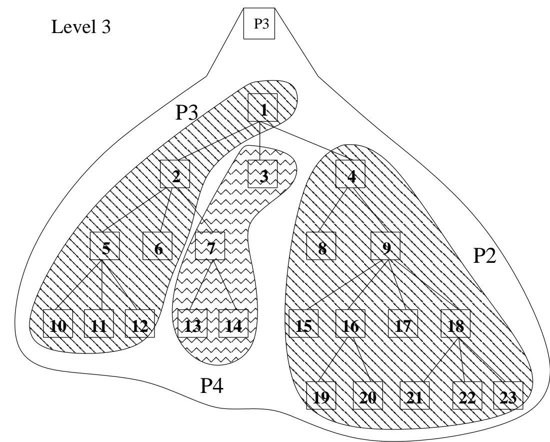


Figure 4: An illustration of load-balancing in D&C. p3 migrates workloads associated with branch node 3 and 4 to idle p4 and p2.

In each of the knowledge modeling, we allow for the expression of algorithm variants that may add load-balancing, task scheduling, or other performance enhancements to a pattern implementation. These introduce new performance knowledge with respect to behavioral models, performance properties, performance-critical design factors, or cause inference in the form of extensions or instantiation of the generic knowledge, as shown in right hand part of Figure 2 . Algorithm-specific knowledge generation can follow the generic pattern-based knowledge extraction approach we presented above. And our knowledge representation, i.e., inference trees, can be readily extended to incorporate the new knowledge as inference branches at appropriate tree levels.

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

4. Case study – Divide-and-Conquer Pattern Knowledge Generation

4.1. Pattern Description

The data-parallel Divide-and-Conquer (D&C) pattern describes a class of parallel programs that features recursively splitting a large problem into a certain number of smaller subproblems of the same type, then solving the subproblems in parallel and merging their solutions to achieve the solution to the original problem [3]. To split the work among the processes,

Copyright © 2005 John Wiley & Sons, Ltd.

all processes first sort out data locally, then exchange data in a specified order. The data redistribution results in two independent process sets whose data are orthogonal (this is due to the nature of divide-and-conquer problems). Each set of processes then continues the process splitting until singular process sets are reached. Thereafter each process can independently work on its assigned data with serial divide-and-conquer code. After the independent computing has finished, processes merge their partial results with sibling processes' to form an overall solution. The pattern behaviors are illustrated in Figure 3. Example problems that can be parallelized with this pattern include Quicksort, Barnes-Hut n-body [21], Delaunay triangulation [22], etc.

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

4.2. Behavioral Modeling

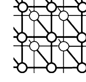
We model the D&C pattern behavior with abstract events depicted in Figure 5. The figure shows a level I event that sketches overall behavior of D&C, and level II events that refine the descriptions of essential pattern components. We display only the full description of event *Solve* in the figure for brevity. The performance attributes are derived from inter-process interactions, which will be used later for synthesizing patter-specific metrics. Other fields of the event description, including constituent event format, constraints, and computing rules of performance attributes are filled in when a concrete algorithmic implementation of the pattern is generated, which we will illustrate in Figure 9.

4.3. Performance Modeling

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

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

Copyright © 2005 John Wiley & Sons, Ltd.



Level I expression

Overall —— (Divide)* ∘ (Solve ∘ Merge)*

Level II expression

Divide —— Local_Divide ∘ Redistribute_Send ∘ Redistribute_Recv
Solve —— [(Find_idle_Proc ∘ Send_MgrLoad) | (Find_busy_Proc∘ Recv_MgrLoad)] ∘ Local_Compute
Merge —— Send2Sibling | (RecvfromSibling ∘ Local_Merge)

Full description of **Solve**

AbstractEvent Solve (id, pid) {
Expression
[(Find_idle_Proc∘ Send_MgrLoad) | (Find_busy_Proc ∘Recv_MgrLoad)] ∘ Local_Compute
Associated Events
Solve BusyProcSolve, IdleProcSolve;
Performance Attributes
Wait_Time4Find_idle_Proc, Wait_Time4Find_busy_Proc, Wait_Time4MgrLoad, Perc_Of_MgrLoad
}

Figure 5: Two-level D&C abstract event descriptions.

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

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

$$t_{comp} = t_{divide} + t_{solve} + t_{merge} \tag{2}$$

$$t_{comm} = t_{comm-divide} + t_{comm-solve} + t_{comm-merge} \tag{3}$$

$$t_{wait} = t_{w-divide} + t_{w-solve} + t_{w-merge} \tag{4}$$

Level II events dictate refined performance models. The time spent in problem solving, for instance, is categorized into three classes, finding an idle process to share workload, finding a busy process to obtain extra workload, and migrating workload, according to the abstract event *Solve* defined in Figure 5.

$$t_{solve} = t_{local_comp} + (t_{comp-find_idle_proc} \ or \ t_{comp-find_busy_proc}) \tag{5}$$

$$\begin{aligned} t_{comm-solve} &= t_{comm-find_idle_proc} + t_{comm-send_mgr_load} \\ & or \quad t_{comm-find_busy_proc} + t_{comm-recv_mgr_load} \end{aligned} \tag{6}$$

$$t_{w-solve} = t_{w-find_idle_proc} \ or \ (t_{w-find_busy_proc} + t_{w-mgr_load}) \tag{7}$$

Copyright © 2005 John Wiley & Sons, Ltd.

4.4. Pattern-specific Metric Definition

The performance models above enable the definition of pattern-specific metrics to use for performance evaluation. Pattern-specific metrics are calculated by categorizing and synthesizing performance attributes in related abstract events. Formulating each item in equation (7), for instance, we get a set of metrics that are associated with load-balancing performance as below:

$$t_{w-find_idle_proc} := \sum_{i=0}^{K} e^{i}_{Wait_Time4Find_idle_Proc} \tag{8}$$

$$t_{w-find_busy_proc} := \sum_{i=0}^{K} e^{i}_{Wait_Time4Find_busy_Proc} \tag{9}$$

$$t_{w-mgr_load} := \sum_{i=0}^{K} e^{i}_{Wait_Time4MgrLoad} \tag{10}$$

where $e^{i}_{Wait_Time4Find_idle_Proc}$, $e^{i}_{Wait_Time4Find_busy_Proc}$, and $e^{i}_{Wait_Time4MgrLoad}$ represent the performance attribute *Wait_Time4Find_idle_Proc*, *Wait_Time4Find_busy_Proc*, and *Wait_Time4MgrLoad* of the ith *Solve* event occurring on the process respectively, and k the number of event instances *Solve*.

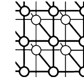
The performance modeling and metric definition in terms of pattern semantics provide a foundation to the generation of inference trees that formally represent the performance problem search and interpretation process.

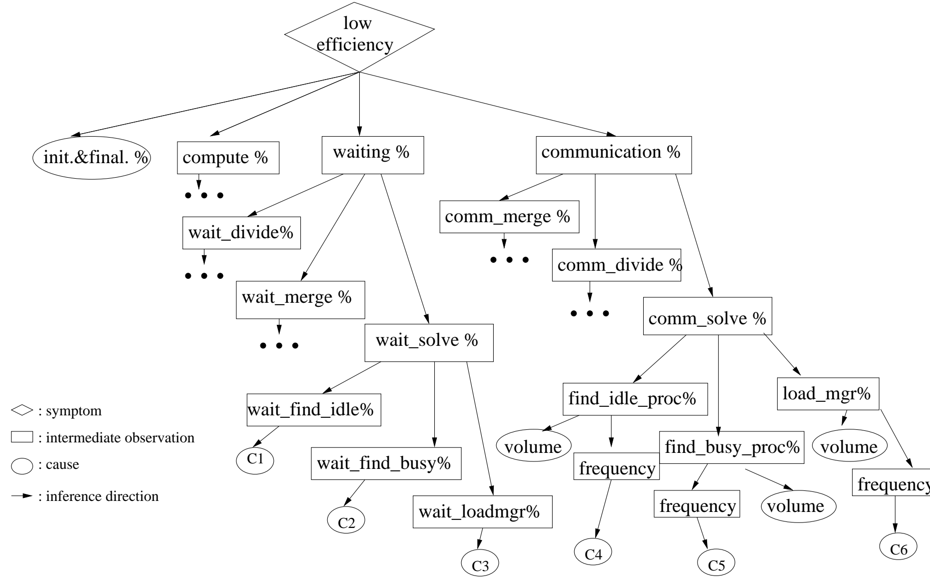
4.5. Inference Modeling

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

- *Concurrency and cost of divide and merge stage*. Due to the nature of D&C problems, it is very likely that not all of the processes are active at divide or merge stage. In the case of low concurrency, if divide or merge operations are expensive, the idle processors will wait a significant amount of time for problem dividing or solution merging, therefore insufficiently utilized.
- *Size of base problem instance*. D&C model is particularly effective when the amount of work required for solving a base case is large compared to the amount of work required for recursive splits and merges. On the other hand, for the purpose of maximizing processor utilization there should be a sufficient number of base problem instances. The two often conflicting conditions give rise to the trade-off between the depth of recursive split and the amount of base problems.
- *Scheduling algorithms used for balancing workload*. The algorithm decides where to migrate workload and the amount of work to be migrated. The factor decides the degree of load balance and communication cost of moving workload around.

Copyright © 2005 John Wiley & Sons, Ltd.





c1: Waiting time for finding a idle process is significant. It is very likely that the load−balancing algorithm is not efficient in locating idle processes.
c2: Waiting time for finding a busy process to migrate its workload is significant. It's likely that the load−balancing algorithm is not efficient in locating busy processes.
c3: Waiting time for the arrival of migrated workload is significant. The possible cause is that deciding workload to be migrated is expensive at the busy processes.
c4: There is a number of communications for finding idle processes to share workload. Change the load−balancing algorithm so that busy processes request help at a moderate frequency.
c5: There is a number of communication for finding busy processes to get workload. One possible cause is that the average workload per migration involves relatively small amount of computation time so that the process requests for extra workload pretty often. Another possible way to improve performance is to change the load−balacing algorithm so that idle processes request extra workload at a moderate frequency.
c6: Load migration is frequent. One possible cause is that the average workload per migration involves relatively small amount of computation time so that the processes request for extrat workload pretty often. Another possible cause is that some load migration trafers too much workload to an idle process so that orignial busy process becomes idle soon and the workload then thrashes between the two processes, causing unnecessary data transfer.

Figure 6: An inference tree for performance diagnosis of Divide-and-Conquer pattern.

The generation of an inference tree of diagnosing D&C programs is based on performance modeling and pattern-specific metric evaluation. An example inference tree for diagnosing a symptom "low-efficiency" is presented in Figure 6. For brevity we present in the figure only inference steps that refer to the performance models and metrics we defined in section 4.3 and 4.4. We can see that inference trees represent a bottom-up performance cause inference approach that links low-level symptoms to causes at high level of abstraction. As inference going deep, causes of performance inefficiency are localized.

The knowledge inference present in Figure 6 is not meant to be complete. A specific implementation of the pattern in an algorithm may introduce new behavioral models and performance factors. Nevertheless, designed to be extensible, our inference trees can readily accommodate the knowledge extension.

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

Copyright © 2005 John Wiley & Sons, Ltd.

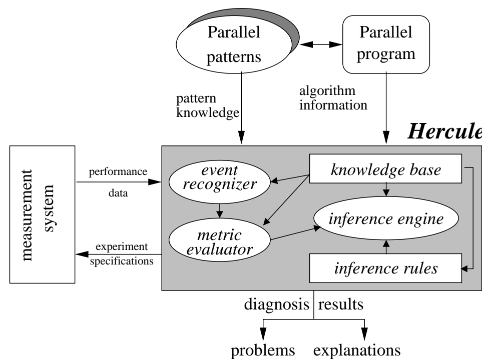


Figure 7: Hercule performance diagnosis framework

```
(defrule <rule-name>
   <condition-element>*
   =>
   <action>*)
where
condition-element (ce) ::= <pattern-ce> |
   <assigned-pattern-ce> | <not-ce> |
   <and-ce> | <or-ce> | <logical-ce> |
   <test-ce> | <exists-ce> | <forall-ce>
action ::= <assertion> | <function call>
```

Figure 8: CLIPS production rule syntax

5. Hercule – A Prototype Automatic Performance Diagnosis System

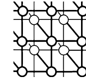
We have built a prototype automatic performance diagnosis system called *Hercule*[†] which implements the pattern-based performance diagnosis approach discussed above; see Figure 7. The Hercule system operates as an expert system within a parallel performance measurement and analysis toolkit, in this case, the TAU [13] performance system. Hercule includes a knowledge base composed of an abstract event library, metrics set, and performance factors for individual parallel patterns. Below, we describe in more detail how the performance knowledge encoding is accomplished in Hercule.

Hercule implements the abstract event representation in a Java class library. This library provides a general programmatic means to capture pattern behaviors, and it allows for algorithm extension. 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 pattern-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 be extended to incorporate algorithm-specific abstract event definitions and metric computing rules.

Perhaps the most interesting part of Hercule is the cause inferencing system. We encode the cause inference processes represented in 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 [11] expert system building tool to describe production rules, and the CLIPS inference engine for operation. The syntax form is shown in Figure 8. The inference engine provided in CLIPS

[†]The name was chosen in the spirit of our earlier performance diagnosis project, *Poirot* [16].

Copyright © 2005 John Wiley & Sons, Ltd.



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.

The effort involved in pattern-based performance knowledge engineering in Hercule consists of two parts: acquiring knowledge with the approach presented in section 3 and encoding the knowledge with abstract event specification, performance metric formulation, and production rules. Work time needed for a performance analyst to generate knowledge varies depending on computational complexity of the pattern and desired detail level of the targeting inference tree. When using the knowledge base to diagnose a parallel application coded with a parallel pattern, 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.

6. Experimentation

In this section, we demonstrate Hercule's ability to diagnose performance problems in a D&C program, parallel Quicksort. Experiments are run on a IBM pSeries 690 SMP cluster with 16 processors. The parallel Quicksort algorithm, using the aforementioned Divide-and-Conquer pattern, first recursively divides processes until such a situation is attained – for any two processes P_i and P_j, if $i < j$ then any data element on P_i is less than or equal to any element on P_j. Then each process independently executes serial version Quicksort. There is no merge stage in the parallel Quicksort. Since a poor choice of pivot in Quicksort may lead to imbalanced data distribution over processes at runtime, we employ a simple runtime load balancing system that uses a centralized manager to manage idle processes and to assign them to busy processes to share work load. Whenever a processor finishes local data sorting, it registers at the manager. Whenever a processor is about to recurse on more than a predefined amount of data in the serial phase, it requests the manager for help. If no idle processor is available, the manager sends back no-help response, and the original busy processor must continue computing on its own. Otherwise, the manager selects an idle processor, and tells it which processor to help and the amount of data to expect. The manager also informs the busy process where to migrate load. The busy processor then migrates the data to the helper, continues with its remaining data load, and waits for the helper to return the result. The actions and relationship of helper (idle) processes, helpee (busy) processes, and the manager are described in the abstract events *Solve, Req2MngEvt, and ManagerEvt* shown in Figure 9. The *Solve* event description instantiated with the Quicksort algorithm is also shown here. Hercule recognizes event instances matching the event specifications as a performance data stream flows through it, calculates the associated performance attributes values, and synthesizes pattern-specific metrics for evaluation in later inference steps.

In Figure 10, Vampir timeline view of a *Solve* phase with load balancing in a Quicksort run with five processes is shown. It also depicts activity chart of the phase. The event trace is generated by the TAU [13] performance measurement system with only major pattern/algorithm components being instrumented. In the figure, dark red regions represent

Copyright © 2005 John Wiley & Sons, Ltd.

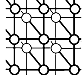


Figure 9: Extended abstract event descriptions of *Solve* in the parallel Quicksort algorithm.

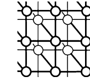
effective computation. Light yellow regions represent MPI function calls, including **MPI_Init**, **MPI_Send**, **MPI_Recv**, etc. In following performance diagnosis, we focus on investigating efficiency of the load-balancing aspect of the Quicksort.

We can see that most of the time is spent in communication (i.e., light yellow areas). It is difficult to tell the communication pattern and what causes the communcations in the event trace stream. However, our abstract event descriptions can fit the seemly intractable performance data into recognizable patterns and help infer performance causes. Hercule diagnosis results of the Quicksort run is shown in Figure 11. Due to limitation of space, only interpretation of solve stage performance is presented here.

Recall that recursive process splitting in D&C pattern leads to a hierarchy of process sets. Each process independently runs sequential D&C code, which can be viewed as a process of expanding and collapsing a *D&C tree* [1] as shown in Figure 4. In the case of load-imbalance (i.e., D&C trees on different processes involve varied workload), some branches of the D&C tree on a process may be migrated to another process which has finished computing. Hercule evaluates and explains performance from the point of view of the D&C trees since the structure makes it easy to interpret inter-process communications for load migrations.

Given the program and performance knowledge associated with D&C pattern, Hercule automatically request an experiment that collects performance event trace for evaluating

Copyright © 2005 John Wiley & Sons, Ltd.



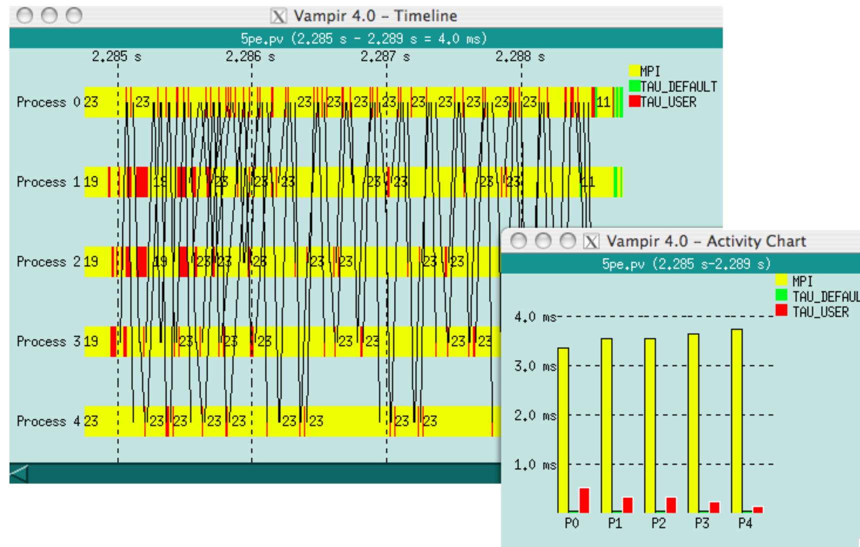


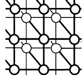
Figure 10: Vampir timeline view in a *Solve* phase of a parallel Quicksort run with five processes.

efficiencies (i.e., computation time/communication time) of each D&C tree at Solve stage. The source code instrumentation of the experiment is specified in accordance with the event expression of Solve in Figure 9. Hercule classifies computation and communication cost and finds that the D&C tree originally rooted at process 2 performs worst. Then Hercule investigates the communication cost associated with D&C tree 2 in detail. Of course, any D&C tree can be identified for additional study. Hercule computes pattern-specific performance metrics and distinguishes load-migration communications and requesting-manager communications, each is presented as the percentage the category contributes to the overall communication time. It then tries to explain the high cost associated with these two types of communications respectively and identifies possible factors giving rise to the communications. The inference process and diagnosis results, as shown in Figure 11, are presented in a manner close to programmer's reasoning and understanding.

7. Related Work

Paradyn [9] 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

Copyright © 2005 John Wiley & Sons, Ltd.




```
dyna6-221:~/PerfDiagnosis/bin lili$ testDC DC.clp 5pe.dup
Begin diagnosing DivideConquer program ... ...
==========================================================================================
Experiment 1 -- generate performance event trace for evaluating performance in Solve stage.
------------------------------------------------------------------------------------------
do experiment 1... ...
------------------------------------------------------------------------------------------
At Solve stage of D&C pattern, each process independently runs sequential D&C code, which
can be viewed as a process of expanding and collapsing a D&C tree. In the case of load-
imbalance, some branches of the D&C tree on a process may be migrated to another process
which has finished computing. We explain performance with respect to the D&C tree structure.

The D&C tree originally rooted at process 2 has the lowest efficiency (computation/communication
ratio), 0.019. Next we look at communication cost associated with the D&C tree 2.
==========================================================================================
Evaluating pattern-specific metrics associated with communication cost of the D&C tree 2.
... ...
------------------------------------------------------------------------------------------
Among the communications with respect to tree 2, moving data to other processes to balance
workload comprises 15.09% of overall communication time, and requesting the manager for
finding an idle process comprises 84.9%.

In averge, every work load migration transfers 89.8% of remaining work to idle processes,
which may cause load imbalance in the subsequent D&C tree computing, thus more communication
due to load migration. In D&C tree 2, work load migrated to other processes is partially
migrated back to the original process that initiates the tree, causing workload thrashing
and unnecessary data transfer.

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

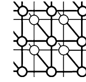
Figure 11: Hercule interpretation of Solve stage performance of a parallel Quicksort run.

targets are not in direct relation to parallel program design. It is not intended for explanation of high-level bug either.

JavaPSL [14] is an implementation of the *Performance Property Specification Language* (ASL) [19] developed by the APART project. 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.

Peridot [20] is an automatic performance analysis tool for teraflop computers. It behaves like a system monitor being able to automatically identify critical applications and to search

Copyright © 2005 John Wiley & Sons, Ltd.



for performance problems in individual applications in a distributed manner. The tool targets applications being programmed in the hybrid programming model combining message-passing and shared-memory programming.

Poirot [16] 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 [17], a Cause-Effect analysis approach is proposed to explain inefficiencies in distributed programs. The approach detects performance losses that 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 [18], 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.

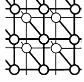
Kappa-Pi [15] is also a rule-based automatic performance analysis tool. In this tool, knowledge about commonly-seen performance problems is encoded into deduction rules at various abstraction levels. It explains the problem found to the user by building an expression of the highest level deduced facts 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 patterns as a basis for performance bug search and explanation. Our method addresses how to extract expert performance knowledge from parallel patterns with four types of modeling: behavioral modeling, performance modeling, pattern-specific metric definition, and inference modeling. We illustrate the knowledge generation approach with the Divide-and-Conquer pattern. Our Hercule framework offers a prototype automatic performance diagnosis system based on the extracted pattern knowledge. We demonstrate the use of Hercule on a Divide-and-Conquer program, parallel Quicksort. Our preliminary results show that pattern-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, Wavefront, and Divide-and-Conquer pattern 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 parallel pattern with the underlying parallel machine. This will require a greater degree of definition in the performance metrics and analysis. Second, parallel applications can use a combination of parallel patterns.

Copyright © 2005 John Wiley & Sons, Ltd.



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 pattern with another. Compositional patterns will naturally include a hierarchical modeling requirement. Finally, we will continue to enhance the Hercule system.

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. SvPablo, University of Illinois, `http://www.renci.unc.edu/Project/SVPablo/SvPabloOverview.htm` [April 10 2005]
3. 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.
4. 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
5. A. Grama, A. Gupta, G. Karypis and V. Kumar. Analytical Modeling of Parallel Programs. In *Introduction to Parallel Computing*. Addison-Wesley, 2003,
6. 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.
7. 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.
8. N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to Perplexed. *ACM Computing Surveys* 1989 21(3):323-357.
9. 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.
10. 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.
11. CLIPS: A Tool for Building Expert Systems, `http://www.ghg.net/clips/CLIPS.html`. [April 10 2005]
12. P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. on Computer Systems* 1995; 13(1):1-31.
13. TAU - Tuning and Analysis Utilities, University of Oregon, `http://www.cs.uoregon.edu/research/paracomp/tau/tautools/` [April 10 2005]
14. T. Fahringer and C. S. Jnio. Modeling and detecting performance problems for distributed and parallel programs with JavaPSL. In *Proceedings of SC2001*, 2001
15. A. Espinosa, Automatic Performance Analysis of Parallel Programs, PhD thesis, Computer Science Department, University Autonoma de Barcelona, Barcelona, Spain, 2000
16. Allen D. Malony and B. Robert Helm, A theory and architecture for automating performance diagnosis. *Future Generation Computer Systems* 2001; 18:189-200.
17. 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
18. Jeffrey Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *ACM International Conference on Supercomputing 2002.*
19. Fahringer, Thomas; Gerndt, Michael; Mohr, Bernd; Wolf, Felix; Riley, Graham. Knowledge Specification for Automatic Performance Analysis APART Technical Report FZJ-ZAM-IB-2001-08, August 2001.
20. Peridot Automatic Performance Analysis for Hitachi SR8000, `https://www.lrr.in.tum.de/~gerndt//peridot/` [April 12 2006]
21. G. Blelloch and G. Narlikar. A practical comparison of N-body algorithms. *Parallel Algorithms, Series inDiscrete Mathematics and Theoretical Computer Science*, vol. 30, 1997.
22. F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Comput. Surv.*, vol. 23, no. 3, pp. 345-405, 1991.

Copyright © 2005 John Wiley & Sons, Ltd.