Model-Based Relative Performance Diagnosis of Wavefront Parallel Computations

Li Li, Allen D. Malony, and Kevin Huck

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

Abstract. Parallel performance diagnosis can be improved with the use of performance knowledge about parallel computation models. The *Hercule* diagnosis system applies model-based methods to automate performance diagnosis processes and explain performance problems from high-level computation semantics. However, Hercule is limited by a single experiment view. Here we introduce the concept of *relative performance diagnosis* and show how it can be integrated in a model-based diagnosis framework. The paper demonstrates the effectiveness of Hercule's approach to relative diagnosis of the well-known Sweep3D application based on a Wavefront model. Relative diagnoses of Sweep3D performance anomalies in strong and weak scaling cases are given.
Keywords: Performance diagnosis, parallel models, wavefront, relative analysis.

1 Introduction

In recent years there has been growing interest in automating the process of parallel performance analysis, including the generation and running of experiments, the comparative analysis of performance results, the characterization of performance properties, and the diagnosis of performance problems. Performance diagnosis is a particularly challenging process to automate because it fundamentally is an intelligent system wherein we capture and apply *knowledge* about performance problems, how to detect them (i.e., their *symptoms*), and why they exist (i.e., their *causes*). Problem discovery and hypothesis testing, as guided by inference-based search, provide the automated reasoning (*explanation*) part of diagnosis automation.

In our work, we focus on performance knowledge engineering as the basis for building a framework to support automated performance diagnosis. The framework's function would be guided by expert strategies for problem discovery and for hypothesis testing, strategies that are captured and encoded in the knowledge base. We advocate looking to models of parallel computations as sources of performance knowledge, which present structural and communication patterns of a program. Models provide semantically rich descriptions that enable better interpretation and understanding of performance behavior. Here, performance

knowledge can be engineered based on the model behavior descriptions so that bottom-up inference of performance causes is effectively supported. A diagnosis system then uses the performance knowledge for performance bug search and reasoning.

We developed the *Hercule* performance diagnosis system [11] to validate how performance knowledge derived from parallel models provides a sound basis for automating performance diagnosis processes and can explain performance loss from high-level computation semantics. This has been shown for several parallel models to date (e.g., master-worker, divide-and-conquer, and domain decomposition). However, we also realized that diagnosis of a single execution is incomplete as a comprehensive diagnosis process. Understanding of performance problems routinely involves *comparative* and *relative* interpretation.

This paper reports our work to improve the Hercule methodology to support what we will term *relative performance diagnosis* (in the spirit of relative debugging [5]). In Section §2, we start with a description of the Hercule framework for diagnosis of the Wavefront model, and then discuss in Section §3 how Hercule is extended to allow relative execution diagnostic analysis. Our target application for study is Sweep3D, mainly for reasons that it is so well-known and understood. We show in Section §4 how Hercule explains certain performance anomalies in strong- and weak-scaling studies. Section §5 highlights related research and Section §6 concludes with observations.

2 Hercule Automatic Performance Diagnosis Framework

Hercule is a prototype automatic performance diagnosis system that implements the model-based performance diagnosis approach. Hercule framework is displayed in figure 1. The Hercule system operates as an expert system within a parallel performance measurement and analysis toolkit, in this case, the TAU [4] performance system. Hercule includes a knowledge base that is composed of an abstract event library, performance metrics set, and performance factors for individual parallel models. The knowledge engineering approach that generates the knowledge base is discussed in [11]. Below, we use the Wavefront model as a driver example to describe Hercule's working mechanism.

Wavefront is a two-dimensional variant of a traditional pipeline pattern. Computation or data is partitioned and distributed on a two-dimensional process grid where every processor receives data from preceding processors and passes down data to successive processors in two orthogonal directions. Well-known pipeline performance problems include sensitivity to load imbalance, processor idleness when pipeline filling up and emptying, and so on. It is these types of problems that we want to find.

We use *abstract events* to describe behavioral and performance characteristics of the Wavefront model. The abstract event describing a Wavefront process node is shown in figure 2. An abstract event description includes constituent primitive events and their format and ordering, a related abstract event list, and associated performance attributes. Hercule implements the abstract event representation in

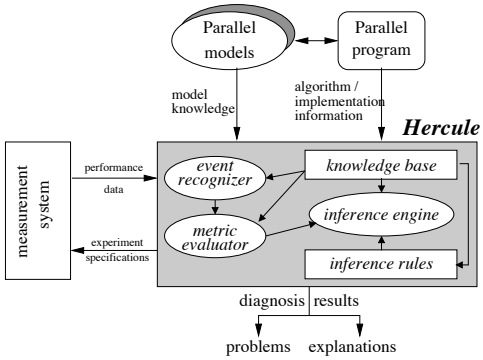

Fig. 1: Hercule performance diagnosis framework

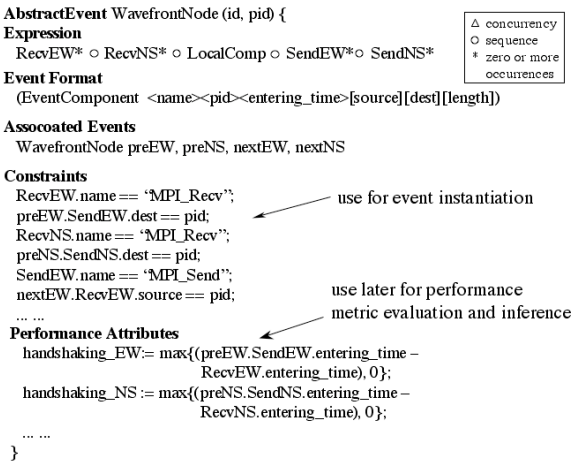

Fig. 2: An abstract event description of Wavefront

a Java class library which provides a general programmatic means to capture model behaviors and allows for algorithm and implementation 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 *metric evaluator*, where performance attributes associated with the event instances are calculated. The metric evaluator also takes in model-specific metric evaluation rules from the knowledge database so it is able to produce metrics by synthesizing performance attributes of the related event instances. Thus, the performance metrics reflect model semantics. A metric named *pl_handshaking* in Wavefront model, for instance, refers to the performance penalty of waiting preceding processes in the Wavefront to pass down data. The model-specific metrics help in mapping low level performance information to a higher level of program abstraction.

Perhaps the most interesting part of Hercule is its cause inferencing system. The expert knowledge used to reason about performance symptoms can be structured as *inference trees* where the root is the symptom to be diagnosed, the branch nodes are intermediate observations obtained so far, and the leaf nodes are high-level performance factors that contribute to the root symptom. An inference tree for diagnosing symptom "low speedup" in Wavefront is presented in figure 3. An intermediate observation is obtained by evaluating a model-specific performance metric against the expected value (from performance modeling) or certain pre-set severity-tolerant threshold. In figure 3, for example, *pl_comm* means the communication cost associated with pipeline message passing. 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.

We encode inference trees with production rules. A production rule consists of one or more performance assertions and performance evidences that must

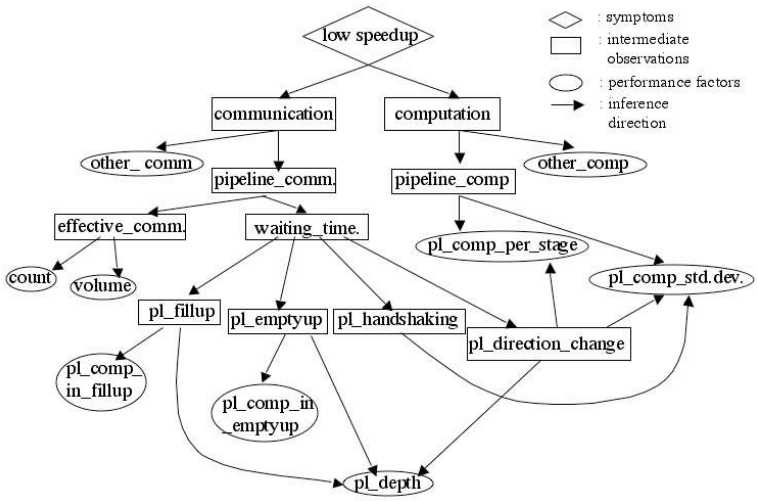

Fig. 3: An inference tree of Wavefront model that diagnoses low-speedup

be satisfied to prove the assertions. Hercule makes use of syntax defined in the CLIPS [2] expert system building tool to describe production rules, and the CLIPS inference engine for operation. The inference engine provided in CLIPS is particularly helpful in performance diagnosis because it can repeatedly fire rules with original and derived performance information until no more new facts can be produced, thereby realizing automatic performance experiment generation and causal reasoning.

3 Hercule Extensions for Relative Performance Diagnosis

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

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

value based on model-specific metrics of the reference run to evaluate problem behaviors. Examples of relative diagnosis of anomalous Wavefront application executions will be presented in the next section.

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

4 Experiment with Sweep3D

In this section, we will demonstrate Hercule's effectiveness in relative performance diagnosis of the ASCI Sweep3D benchmark which uses a Wavefront computational model. Sweep3D [1] is a solver for the 3-D, time-independent, neutron particle transport equation on an orthogonal mesh. Its parallelism comes from multiple wavefronts in multiple dimensions, which are partitioned and pipelined on a distributed memory system. The three-dimensional space is partitioned on a two-dimensional processor grid, where each processor is assigned one columnar domain. Sweep3D exchanges messages between adjacent processors in the grid as wavefront propagates diagonally across the 3-D space in eight directions. Sweep3D is a well-researched parallel benchmark. Although parallelism overheads in Sweep3D have been minimized, for instance, by evenly distributing data across a process grid, leaving little room for performance tuning, Hercule can tell exactly how running time is spent in terms of model semantics, helping understand inherent performance losses of the model under an optimistic condition. Our performance study with Sweep3D focuses on overall scalability, looking at how well the application scales as the number of processors is increased (strong scaling) and as total problem size increases with the process count increase (weak scaling).

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

4.1 Case I: Diagnose strong scaling performance problems

Figure 4 shows the strong scaling behavior of Sweep3D with problem size 150^3, and angle blocking factor, *mmi*, equal to 3, k-blocking factor, *mk*, equal to 10. The application scales well in general, but at process count 32 the speedup drops and bounces up when process count increases to 36. We applied Hercule to contrast performance of run1 (with 32 processors) against run2 (with 36 processors) and diagnose performance anomaly cause. Hercule uses relative speedup (compared

to two-processor run) to evaluate performance since there is no inter-processor communication in a sequential execution. The results that follow were generated in a completely automated manner.

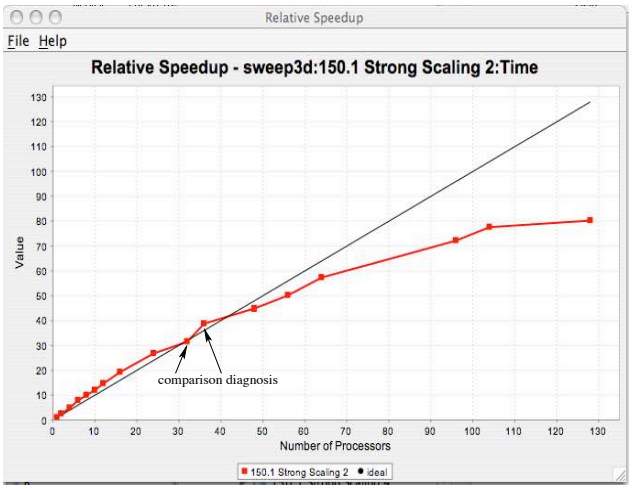

Fig. 4: Sweep3D strong scaling with problem size 150x150x150 (mmi=3, mk=10)

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

Hercule diagnosis step 1: find performance symptom

```
dyna6-166:~/PerfDiagnosis lili$ ./model_diag WF_speedup.clp 150.32pe.dup 150.36pe.dup 150.2pe.dup
Begin diagnosing ...
================================================================================
Speedup of run1 and run2 relative to run3
               run1        run2       expected run1
speedup        12.80       15.84      14.08
--------------------------------------------------------------------------------
run1 is slower than the expected value 14.08
--------------------------------------------------------------------------------
Next we look at the symptom low speedup.
================================================================================
```

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

Hercule diagnosis step 2: locate poorly performed functional groups

```
================================================================================
Level 1 experiment -- generate performance data with respect to computation and communication.
--------------------------------------------------------------------------------
Relative speedup of functional groups in run1 and run2
                        run1            run2          expected run 1
computation:            16.035          19.906           17.694
communication:          1.115           1.172            1.042
```

```
-----------------------------------------------------------------------------------------
computation in run1 is longer than the expected.
-----------------------------------------------------------------------------------------
Next look at performance with respect to pipeline components.
=========================================================================================
```

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

Hercule diagnosis step 3: refine locating poorly performed functional groups

```
=========================================================================================
Level 2 experiment -- generate performance data with respect to pipeline components.
-----------------------------------------------------------------------------------------
Relative speedup of pipeline components in run1 and run2
                             run1         run2      expected run 1
computation in pipeline:     16.598       20.702      18.402
other computation:           10.452       12.405      11.03
-----------------------------------------------------------------------------------------
computation in pipeline in run1 is slower than the expected most.
-----------------------------------------------------------------------------------------
Next look at computation in pipeline.
=========================================================================================
```

Since pipeline computation per process in run1 is more expensive than the expected, Hercule then looks at how well the pipeline computation is distributed on all processes.

Hercule diagnosis step 4: form performance hypothesis

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

-----------------------------------------------------------------------------------------
Standard deviation of pipeline computation in run1 is significantly larger than run2, which
implies a load imbalance across processes.
-----------------------------------------------------------------------------------------
Next testify the hypothesis load imbalance.
=========================================================================================
```

Hercule forms a load imbalance performance hypothesis based on the standard deviation of pipeline computations on all processes. Hercule testifies the hypothesis by looking at pipeline model-related parallelism overheads to which load imbalance possibly contributes most. It calculates and distinguishes performance impact of load imbalance on the overhead categories, and exemplifies occurrence of load imbalance with process behaviors in some specific computation step (iteration) and pipeline sweep. This way of explanation provides the users with both the nature of performance causes and evaluations of performance impact of the causes.

Hercule diagnosis step 5: testify performance hypothesis by evaluating pipeline overheads

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

```
Passing along data among successive pipeline stages (handshaking) takes 14.9% of pipeline
communication time. Pipeline handshake delay is unevenly distributed across processes.
std dev = 486463.75. process 31 involves the longest pipeline handshake cost.
--------------------------------------------------------------------------------
Level 3 experiment for diagnosing handshaking related problems -- collect performance event
trace with respect to process 31
--------------------------------------------------------------------------------
Pipeline HS delay is evenly distributed across iterations in the process 31. Next we look at
performance characteristics of iteration 3 which involves the longest pipeline HS.

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

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

In general, process 31 is assigned 23.6% less work load than process 4. Such discrepancy
causes process 31 idle for 29.5% of pipeline communication time..
--------------------------------------------------------------------------------
When pipeline sweep direction change, processes may be idle waiting for successive pipeline
stages in previous sweep to finish up, and for pipeline to fill up in a new sweep.  The sweep
direction changes comprise 34.6% of pipeline communication time. The delay is unevenly
distributed across processes. process 31 involves the longest pipeline direction change cost.
--------------------------------------------------------------------------------
Level 3 experiment for diagnosing sweep-direction-change related problems -- collect
performance event  trace with respect to process 31
--------------------------------------------------------------------------------
Pipeline direction change delay of process 31 is unevenly distributed across iterations. Next
we look at performance of the iteration 10 which involves the longest direction change delay.

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

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

4.2 Case II: Diagnose weak scaling performance problems

The second experiment with Hercule demonstrates its capability of identifying and explaining parallelism overhead increases as both problem size and process count are increased in weak scaling study. Figure 5 shows the weak scaling behavior of Sweep3D with fixed problem size 20x20x320. We can see that runtime increases as more processors are used even though each process's computation load is kept the same. Hercule will compare 4-processor and 48-processor run and report and explain the performance difference. Again, the results that follow are generated in a completely automated fashion.

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

Hercule diagnosis step 1: find performance symptom

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

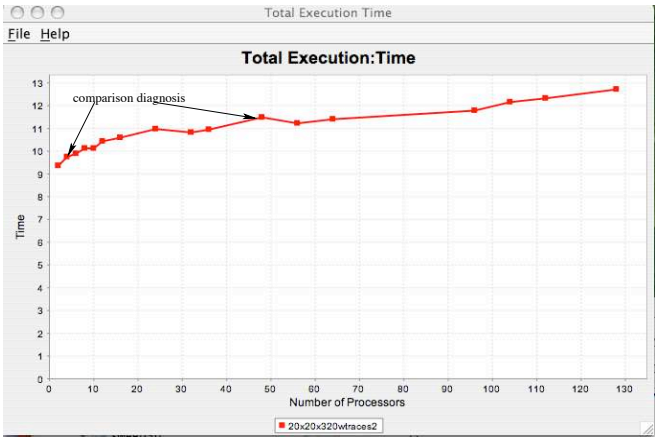

Fig. 5: Sweep3D weak scaling with fixed problem size 20x20x320 per processor (mmi=3, mk=10)

```
Begin diagnosing ...
================================================================================================
Runtime of run1 and run2 (in seconds)
                        run1        run2       difference%
runtime                11.489      9.815        17.055%
------------------------------------------------------------------------------------------------
run1 is 17.055% slower than the run2.
------------------------------------------------------------------------------------------------
Next we look at the symptom parallelism overhead.
================================================================================================
```

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

Hercule diagnosis step 2: locate poorly performed functional groups

```
================================================================================================
Level 1 experiment -- generate performance data with respect to computation and communication.
------------------------------------------------------------------------------------------------
Runtime of functional groups in run1 and run2 (in seconds)
                         run1            run2          difference%
computation:             8.886           8.891         -5.624e-4
communication:           2.603           0.924         181.71%
------------------------------------------------------------------------------------------------
communication cost in run1 is significantly higher than run2.
------------------------------------------------------------------------------------------------
Next look at communication performance with respect to pipeline components.
================================================================================================
```

Hercule further distinguishes pipeline-related communication and others.

Hercule diagnosis step 3: refine locating poorly performed functional groups

```
================================================================================================
Level 2 experiment -- generate performance data with respect to pipeline components.
------------------------------------------------------------------------------------------------
Runtime of pipeline in run1 and run2 (in seconds)
                                                  run1        run2        difference%
computation in pipeline:                         8.014       8.013       1.25e-4
other computation:                               0.872       0.878       -6.83e-3
```

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

comm. count in pipeline (count/per process):        12288       12288         0
comm. volume in pipeline (byte/per process):        58982400    58982400      0
-----------------------------------------------------------------------------------------------
waiting time in pipeline in run1 is 476.62% higher than run2.
-----------------------------------------------------------------------------------------------
Next look at pipeline overheads.
===============================================================================================
```

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

Hercule diagnosis step 4: locate poorly performed pipeline components

```
===============================================================================================
Level 3 experiment -- generate performance data with respect to pipeline waiting time
-----------------------------------------------------------------------------------------------
Runtime of pipeline components in run1 and run2 (in seconds)
                                              run1            run2          difference%
 waiting time in pipeline                     1.332           0.231          476.62%
 pipeline fill-up:                            0.161           0.014          1050%
 pipeline empty-up:                           0.244           0.017          1335.29%
 pipeline handshaking:                        0.337           0.075          349.33%
 pipeline direction change:                   0.584           0.125          367.2%
===============================================================================================
```

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

Hercule diagnosis step 5: diagnose two most pronounced pipeline overheads

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

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

In iteration 0, the depth of pipeline is 13. The pipeline tail, process 0 is being idle while
the pipeline is filling up by processes in preceding stages. The pipeline fill-up delay
comprises 335103us (20.8%) of process 0's total waiting time. The computations at preceding
pipeline stages together account for the long waiting time at the process. Reducing
computation load at preceding stages or pipeline depth will decrease filling up time.

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

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

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

In iteration 4, the depth of pipeline is 13. The pipeline head, process 0 is being idle while
the pipeline is emptying up by processes in successive stages. The pipeline empty-up
delay comprises 573162us (35.5%) of process 0's total waiting time. The computations at
successive pipeline stages together account for the long waiting time at the process. Reducing
```

```
computation load at successive pipeline stages or pipeline depth will decrease empty-up time.

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

In iteration 1, the depth of pipeline is 3. The pipeline head, process 0 is being idle while
the pipeline is emptying up by processes in successive stages. The pipeline empty-up
delay comprises 34858us (31.0%) of process 0's total waiting time.
================================================================================
```

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

5 Related Research

Our work draws inspiration from important parallel performance analysis projects and is one of several that are pursuing automated performance diagnosis. There are two main aspects of the work to discuss in relation to other research: 1) knowledge-based diagnosis automation, and 2) multi-experiment comparative and relative analysis.

The Paradyn [12, 13] and Kappa-PI [6] projects are the closest to our work on Hercule in terms of knowledge-based problem discovery. Paradyn is a performance analysis system that automatically locates bottlenecks using the W^3 search model. According to the model, searching for a performance problem is an iterative process of refining the answer to three questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. This is a powerful methodology for problem search [13, 14] and diagnosis automation, but to a lesser extent than Hercule, the performance bugs Paradyn targets are not in direct relation to parallel program design and provide less explanation of high-level causes. Kappa-Pi is another automatic performance diagnosis tool that encodes knowledge about commonly-seen performance problems into deduction rules at various abstraction levels. It explains the problem found by building an expression of the highest-level deduced fact which includes the situation found, the importance of such a problem, and the program elements involved in the problem. Our work builds on the Kappa-Pi objectives by proposing a systematic approach to extracting knowledge from high-level parallel design patterns. It should be noted that the research represented by the Kojak project [3] also supports performance problem search as guided by event patterns found in parallel execution traces.

The work by Karavanic [9, 10] on comparative, multi-experiment performance analysis makes a strong case for applying such capabilities to diagnosing performance problems that manifest themselves differentially. The resource hierarchies and other experiment information allows an exploration of a performance space

that can provide semantic context for diagnosis explanation. In a similar spirit, the performance algebra work by Wolf [15] allows for comparision, integration, and summarization of performance across multiple experiments. What distinguishes this research is its usage of a data model to represent multi-dimensional performance information, including performance problems detected for each experiment, and the algebraic manipulation of the data to discover new performance facts. Multi-experiment performance analysis is best supported by infrastructure for maintaining performance information. The work on PerfTrack [8] and PerfExplorer [7] are important projects in this regard.

6 Conclusions

The use of relative performance diagnosis, where problems are discovered and explained in relation to other experiments, is important to support in diagnosis systems. For model-based diagnosis frameworks such as Hercule, we look to integrate relative analysis in the knowledge and inference engineering. In this paper, we report on how the Hercule framework was extended to enable relative diagnosis by adding an interface from the metric evaluator to the inference engine to analyze performance from different runs. In addition, decision rules were developed for problem identification and hypothesis testification. The paper demonstrates the effectiveness of the approach to relative diagnosis of the well-known Sweep3D application based on a wavefront model. Relative diagnoses of performance anomalies in strong and weak scaling cases are given.

References

1. The asci sweep3d benchmark. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/`.
2. Clips: A tool for building expert systems. `http://www.ghg.net/clips/CLIPS.html`.
3. Kojak – kit for objective judgement and knowledge-based detection of performance bottlenecks. `http://www.fz-juelich.de/zam/kojak/`.
4. Tau tuning and analysis utilities. `http://www.cs.uoregon.edu/research/paracomp/tau/tautools/`.
5. D. Abramson, I. Foster, J. Michalakes, and R. Sosic. Relative debugging: A new paradigm for debugging scientific applications. *The Communications of the Association for Computing Machinery*, 39(11):67–77, 1996.
6. A. Espinosa. *Automatic Performance Analysis of Parallel Programs*. PhD thesis, Computer Science Department, University Autonoma de Barcelona, Barcelona, Spain, 2000.
7. K. Huck and A. Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *SC 2005*, 2005.
8. K. Karavanic, J. May, K. Mohror, B. Miller, K. Huck, R. Knapp, and B. Pugh. Integrating database technology with comparison-based parallel performance diagnosis: The perftrack performance experiment management tool. In *SC 2005*, 2005.

9. K. Karavanic and B. Miller. Improving online performance diagnosis by the use of historical. In *SC'99*, 1999.
10. K. Karavanic and B. Miller. A framework for multi-execution performance tuning. In *On-line Monitoring Systems and Computer Tool Interoperability*, pages 61–89. Nova Science Publishers, Inc., 2004.
11. L. Li and A. Malony. Knowledge engineering for automatic parallel performance diagnosis. Accepted by *Concurrency and Computation: Practice and Experience*.
12. B. Miller and et al. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.
13. P. Roth and B. Miller. Deep start: A hybrid strategy for automated performance. In *EuroPar 2002*, 2002.
14. P. Roth and B. Miller. On-line automated performance diagnosis on thousands of processes. In *PPoPP 2006*, 2006.
15. F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *ICPP 2004*, 2004.