Automatic Performance Diagnosis of Parallel Computations with Compositional Models

Li Li and Allen D. Malony

Performance Research Laboratory
Department of Computer and Information Science
University of Oregon, Eugene, Oregon 97403
{lili, malony}@cs.uoregon.edu

Abstract

Performance tuning involves a diagnostic process to locate and explain sources of program inefficiency. A performance diagnosis system can leverage knowledge of performance causes and symptoms that come from expertise with parallel computational models. This paper extends our model-based performance diagnosis approach to programs with multiple models. We study two types of model compositions (nesting and restructuring) and demonstrate how the Hercule performance diagnosis framework can automatically discover and interpret performance problems due to model nesting in the FLASH application.

1 Introduction

There is a growing interest in automating the process of parallel performance analysis, from measure generation to performance diagnosis. *Performance diagnosis* is a particularly challenging process to automate because it fundamentally is an intelligent system wherein we capture and apply *knowledge* about performance problem detection, and explanation about why it exist. Problem discovery and hypothesis testing, as guided by inference-based search, provides the automated reasoning (*explanation*) part of diagnosis automation, if only we had a sound basis for performance knowledge engineering. In our work, we advocate *models* of parallel computations as sources of performance knowledge. Models provide semantically rich descriptions of structural, control flow, and communication patterns of a program, enabling better interpretation and understanding of performance properties. Performance knowledge can be systematically engineered based on the model behavior descriptions and bottom-up inference methods used to encode model-specific expert strategies for problem discovery.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

In the previous work [7, 8, 9], we focused on generating and encoding performance knowledge from *singleton* models, including *master-worker*, *pipeline*, and *divide-and-conquer* models, and building the Hercule performance diagnosis framework to support these singleton models. However, scientific programmers often combine two or more computational models in parallel applications. *Compositional* models capture how singleton models are composed together and interact in a parallel program. Here we report our work on performance diagnosis of compositional models. We present an approach for discovering and interpreting performance bugs using both the semantics of individual models and their composition properties. We extended our Hercule performance diagnosis framework to support performance engineering of compositional models and have tested Hercule on the scientific application FLASH [3] and the ScaLAPACK algorithm PDLAHQR [4], each representing different types of model integration. These experiences demonstrate that our approach can effectively support automatic diagnosis of compositional model performance.

2 Computational Model Composition

A *parallel computational model* [10, 11], also called a *parallel pattern* or *programming paradigm*, is a recurring parallel solution to a class of problems. *master-worker*, *pipeline*, and *geometric decomposition* are well-known models. Models are useful because they abstract parallel execution details and provide a semantic basis for parallel program development. In previous work [7, 8], we investigated how to generate and encode performance knowledge from single models to support automated performance diagnosis. This work resulted in the development of the Hercule performance diagnosis framework. While the results using Hercule were generally successful, real-world applications are more complex, often based on the composition or synthesis of two or more elementary computational models. To conduct performance diagnosis of a compositional parallel program we must extend the knowledge engineering and problem inferencing to capture the interplay of one model with another.

Consider a parallel computational model as a set of indivisible computational components, $\{C_1, C_2, ..., C_k\}$, and a function, $F(C_1, C_2, ..., C_k)$, that specifies the relative control order (e.g., sequential, choice, concurrent, iteration, and so on) of component occurrence. We can then regard the composition of two models, $F(...)$ and $G(...)$, as an integration of the components in some manner. Several compositional forms are possible. For instance, one model could simply nest one model hierarchically within another (model nesting), or the component sets of two models could be restructured in a more complex way by a higher-order function (model restructuring). Our objective is to understand the compositional properties of model integration in order to engineer the performance knowledge needed for performance diagnosis. Our approach will describe how performance effects of individual models change as the components merge and how new performance effects arise from the composite interactions.

2.1 Model Nesting

An important class of parallel composition uses a high-level *root* (outer) model to describe high-level parallel behavior and lower-level *child* (inner) models to describe parallelism within the root model's components. We call this type of composition *model nesting*. Stated more formally, two models $F(C_1, C_2, ..., C_k)$ and $G(D_1, D_2, ..., D_l)$ (D_i are components of G) may compose into a new nested model as follows:

$$\begin{aligned} &F(C_1, C_2, ..., C_k) + G(D_1, D_2, ..., D_l) \rightarrow \\ &\quad F(C_1\{G(D_1, D_2, ..., D_l)\}, \\ &\qquad C_2\{G(D_1, D_2, ..., D_l)\}, \\ &\qquad \cdots, \\ &\qquad C_k\{G(D_1, D_2, ..., D_l)\}) \end{aligned} \tag{1}$$

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

Parallel applications based on nested computational models are common. Iterative, multi-phase applications are frequently structured as nested models with an outer code controlling multiple phases each based on a possibly different parallel pattern. The graphical animation described in [6] implements expensive pipeline stages with master-worker model. The FLASH [3] code we will study later nests parallel recursive tree computations in an adaptive mesh refinement model. Our concern is how to understand the performance of nested models. Due to the hierarchical structure of model nesting, analysis this type of application usually starts with the root model. When a problematic component is found in the model (e.g., an expensive phase), we switch from the root to the component's model to refine performance problem search. The search continues until the finest level of model is reached. Performance overhead categories of the nested model is the union of the overheads associated with the participant models. Thus, they should be organized in a hierarchy conforming to the model nesting structure to support the top-down bug search.

2.2 Model Restructuring

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

$$\begin{aligned} &F(C_1, C_2, ..., C_k) + G(D_1, D_2, ..., D_l) \rightarrow \\ &\quad H((\{C_1^F, ..., C_k^F\} \mid \{D_1^G, ..., D_l^G\})^+) \end{aligned} \tag{2}$$

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

The general idea is that the components of the contributing models are being mixed to form a new set, to which a new model function H is applied. H could be F, G, or a new operation, like iteration, nesting, farming, and so on. A simple example might be the restructuring of two pipeline models into a single pipeline model with the components at each pipeline stage merged. More complex examples are the nonsymmetric QR algorithm (PDLAHQR) [4] in ScaLAPACK, which combines pipeline and geometric decomposition models, and the MUMPS sparse direct solver [5], where parallel tree, master-worker, and geometric decomposition models are mixed together.

For our purposes, the key difference between model nesting and model restructuring has to do with the notion of *working context*. In model restructuring, the working context of a component from a contributing model will be different from its context in the singleton form of that model. The performance overheads associated with the original models (see [7, 9]), will change corresponding to context-specific factors and the new model function H. In contrast, when computational models are nested, the model semantics at each level of hierarchy will be preserved, and the working context for components of a nested model will be model-local. From a performance diagnosis perspective, the performance overheads and problem causes can thus be isolated to the models used at different levels of hierarchy.

Diagnosing a parallel program based on model restructuring requires we learn how performance effects of individual models change as the components are interleaved. New performance characteristics introduced by model restructure include *delegated delay* and *composite delay*. When components of a model are intermixed with those of another model, the performance delay associated with a component are manifested within another model components's execution context. We call the performance delay the *delegated delay*. From Equation 2, a delay originally caused by C_i^F and manifested in a later component C_j^F, may now appear in another component, say C_k^G occurring before C_j^F. This delay is then an example of delegated delay and should be attributed to its original model.

A delay jointly caused by two or more models is a *composite delay*. Here, a performance delay associated with an original model manifests itself as part of delays from other interleaved models. From Equation 2, a performance delay happening inside C_i^F, may reflect the cumulative effects of preceding F and G components, as its working context switches from solely F (in the singleton model) to mixed F and G. To assess composite delays, we need to change original performance evaluation rules in response to the model interaction. In the next section, we will discuss how to incorporate the compositional performance effects into the automatic diagnosis framework.

3 Performance Knowledge Engineering Adaptive to Model Composition

At the core of our automatic performance diagnosis approach is engineering performance knowledge from computational models. This proceeds in four stages: behavioral modeling $\rightarrow$ performance modeling $\rightarrow$ model-specific metric definition $\rightarrow$ inference modeling [8]. The model-specific knowledge is stored into a knowledge base which interfaces to an inference engine to drive performance diagnosis search. Base models may be customized by a parallel program and introduce new diagnostic requirements as to problem discovery and inferencing. The user can follow the engineering principles to extract adapted knowledge step by step and then join them with the inherited model knowledge. Model composition is another type of model variation. In this section, we describe how the knowledge engineering approach is extended to address performance issues arising from the model interaction. A user can use the approach as the guideline to "compose" knowledge about a compositional model from already available knowledge of participant models.

3.1 Behavioral Modeling

Behavioral modeling captures program execution semantics as behavioral models represented by a set of *abstract events* at varying detail levels, depending on the complexity of the model and diagnosis needs. The purpose of the abstract events in the diagnosis system is to give contextual informaton for performance modeling, metric definition, and diagnostic inferencing. An abstract event description essentially includes an *expression* that takes the form of $F(C_1, C_2, ..., C_k)$. The expression names the component C_i (typically, an indivisible computational component or communication function) and specifies the control ordering F using event operators. Model composition may interleave components of abstract events from different models.

We describe behavioral characteristics of a compositional model (termed *composite events*) by integrating already available abstract events of the participant models (termed *basic events*) in a manner that conforms to their composition style (e.g., nest or restructure). We use the order operators *sequential* ($\circ$), *choice* ($|$), *repetition* (+ or *), and *occur zero or one time* ([]) to specify occurrence (control) order of the basic events. For instance, model nesting (see Equation 1) requires that a component in the root model (a basic root event) be replaced by the whole set of basic events from the child model (in fact, the whole child model).

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

3.2 Performance Modeling

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

Performance metrics in a compositional model are not simply a union of the metrics in participant models. They may change as to their occurrence locations and evaluation rules. In restructured models, delegated delay and composite delay are important to identify in performance models for composite abstract events. The annotated components at model switch points provide a clue to where a performance delays possibly transfer. A performance loss that originally happens in a model now should take into account the interleaving model's cumulative effects in the evaluation.

3.3 Inference Modeling

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

We generate the inference tree for a compositional model by merging the inference trees of the participant models. The tree merge for a nested model is based on its model hierarchy, where we expand the inference tree of the root model with relevant tree branches of the child models in the hierarchy. Recall that each node in an inference tree represents an intermediate observations that is obtained by evaluating a model-specific metric. If an involving component in a metric evaluation is refined by a new model, the sub-trees associated with the metric will be expanded with the new model's inference tree. An example of merging inference trees for model nest is presented in section 5.3.

For constructing an inference tree of model restructuring, we pick one model tree as the host to expand with inference processes of a second model. The host tree is usually the one of highest complexity among the involving models. We add in the inference tree of the second model node by node – we look for the node's correct location in the host and build its connections with the host tree nodes according to the interaction pattern of the two models. Starting from the root node, if there is a node in the host that represents the same semantics (i.e., performance metric type), we remove the node from the second tree and set its equivalent node in the host as parent of its sub-trees. Otherwise, we remove the node and its sub-trees from the second tree and merge them under the node's parent in the host. In this case, we also need to check if the node or its children represents a performance metric that is transfered from the host due to the model interaction (i.e., delegated performance metric), or vice versa. We draw a line pointing from the deputy model node to the delegator model node. Similarly at a node representing a composite metric, relevant nodes from the both models will connect to the node to reflect the cumulative performance effect. The merge continues until the second tree is empty.

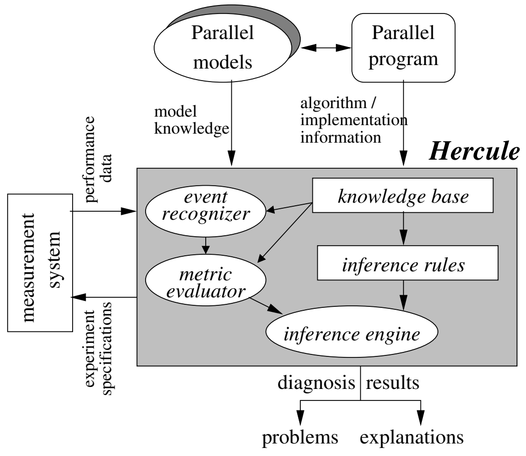

Figure 1. Hercule diagnosis framework

4 Hercule Automatic Performance Diagnosis Framework

Hercule is a prototype automatic performance diagnosis system that implements the model-based diagnosis approach. The Hercule framework, shown in figure 1, operates as an expert system within a parallel performance measurement and analysis toolkit, in this case, the TAU [2] performance system. Hercule includes a knowledge base composed of an abstract event library, performance metrics set, and performance factors for individual parallel models. 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 synthesized and calculated into metrics according to model-specific metric evaluation rules from the knowledge database.

Perhaps the most interesting part of Hercule is its cause inferencing system. We encode inference trees with production rules. A production rule consists of one or more performance assertions and performance evidences that must be satisfied to prove the assertions. Hercule makes use of syntax defined in the CLIPS [1] expert system building tool to describe production rules, and the CLIPS inference engine for operation. The inference engine provided in CLIPS is particularly helpful in performance diagnosis because it can repeatedly fire rules with original and derived performance information until no new facts can be produced, thereby realizing automatic performance experiment generation and causal reasoning.

Hercule's singleton model analysis facilities can also support the compositional diagnosis if provided with the performance knowledge specific to the model composition. Given two models whose performance knowledge has been stored in the knowledge base, the user needs to generate and input the extra knowledge imposed by their interaction pattern to diagnose a specific algorithm, which includes the combined abstract event descriptions, composite metric evaluation rules, performance factors specific to the model interaction, and interfacing inference steps that link two inference trees together in accordance with their interaction pattern. The guidelines to generate the compositional knowledge from the already available base model knowledge have been provided in section 3. The knowledge engineering approach, in contrast to building everything from the scratch, can effectively reduce the users' burden enforced by the diagnostic process.

5 Experience with FLASH

FLASH [3] is an astrophysical hydrodynamics code developed at the center for Astrophysical Thermonuclear Flashes at the University of Chicago. FLASH is intended for parallel simulations that solve the compressible Euler equations on an block-structured adaptive mesh. Adaptive Mesh Refinement (AMR) is handled using the PARAMESH library, which employs a tree structure of logically Cartesian blocks to cover the computational domain. Each block in the domain is refined by halving the block along each dimension and generating a set of new sub-blocks, each of which has a resolution twice that of the parent block. When a block is de-refined, sibling blocks are removed. Each block is represented by a node in the tree structure. The node stores information about its parent block, child blocks, and neighboring blocks. Load balancing is accomplished by redistributing blocks using a Morton space-filling curve, after all refinements and de-refinements are completed. So a block may be placed on a different processor from it parent or siblings.

AMR dictates a set of basic operations, including guardcell filling, refining and de-refining grids, prolongation of the solution to newly created leaf blocks, and so on. In the FLASH implementation, implied in the operations is the communications required by the grid block tree structure with the blocks being distributed to different processors and the maintenance of the tree structure with mesh refinement and de-refinement. We therefore view the FLASH code as a combination of two parallel computational models, AMR and Parallel Recursive Tree (PRT).

5.1 Identify and Describe Model Composition Pattern

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

- *AMR_Refinement* – refine a mesh grid
- *AMR_Derefinement* – coarsen a mesh grid
- *AMR_LoadBalance* – even out workload among processors after a refinement or derefinement

The data operations corresponding to the mesh rebuilding include:

- *AMR_Guardcell* – update guard cells at the boundary of each grid block with data from the neighbors
- *AMR_Prolongation* – prolong the solution to newly created leaf blocks after refinement
- *AMR_Restriction* – restrict the solution up the block tree after derefinement
- *AMR_DataRedistribution* – data redistribution resulting from mesh redistribution when balancing workload

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

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

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

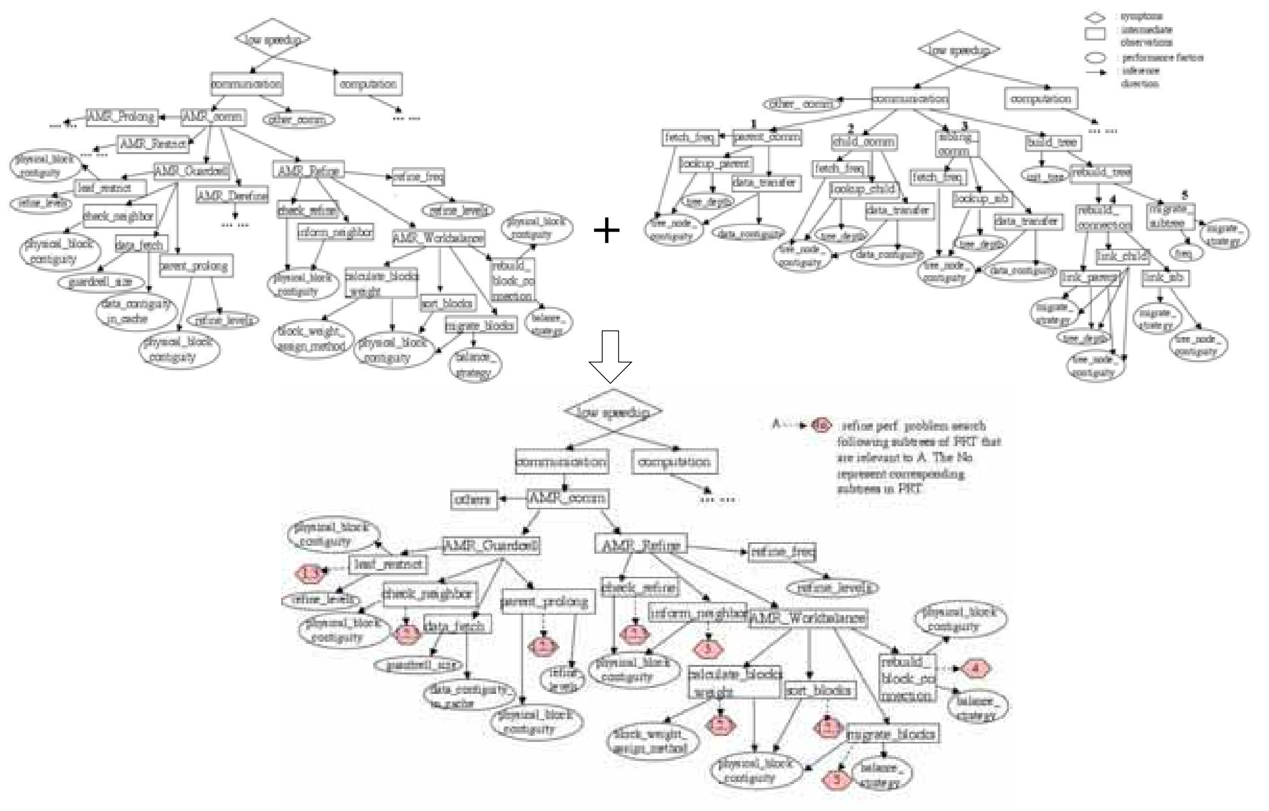

Figure 2. Merge inference trees for FLASH. The top two trees represent AMR and PRT performance inference, and they combine into the FLASH inference tree on the bottom according to the model nest in the FLASH code. Added PRT subtrees are highlighted in the FLASH tree and marked with indices in their original PRT tree. Some subtrees are abbreviated for conciseness.

5.2 Performance Modeling and Metrics

As a nested model, we intend to explain the performance of the FLASH code in terms of its model hierarchy. The performance modeling starts with the root AMR model, then fills the lower-level PRT model into the root framework to refine performance overhead categories.[1] Performance overhead types of the nested model composition is effectively the union of the overheads associated with the individual participant models. To reflect the model hierarchy structure, however, we refine a performance overhead further into a number of context-aware types that indicate different occurrence circumstances within the model heirarchy. Guardcell filling, for instance, is one of the main sources of program inefficiency in AMR model. Using PRT for data communication, the guardcell filling overhead can be further broken down in terms of PRT overhead classes (i.e., communication cost with the parent, child, and neighbors nodes in the grid block tree). Performance metrics, derived from the overheads, are also organized according to model hierarchy, to support the top-down performance problem search. In this way, we allow for capturing performance bugs at different model levels and provide a context for performance explanation in terms of the inter-level model integration.

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

5.3 Merging Inference Trees

To construct the FLASH inference tree, we essentially extend the AMR inference tree with PRT inference steps. Figure 2 shows the merging process. We can see from the figure that some AMR subtrees are extended with PRT branches, which means that a performance problem found relating to the subtree roots can be further tracked down to the PRT operations used. *parent_prolong* in AMR guardcell filling, for instance, involves communications to parent, child, and/or sibling in the grid tree. Its performance counts on these PRT operations along with the factors in the original AMR model. So when there is a performance problem

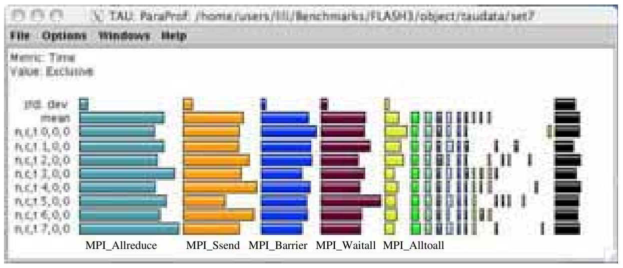

Figure 3. Paraprof view of performance profiles in the FLASH run.

with *parent_prolong*, we should incorporate the relevant PRT inference steps to find the causes.

5.4 Diagnosis Results

We experimented with the Sedov explosion simulation in the FLASH v3.0[2], and ran the simulation on a IBM pSeries 690 SMP cluster with eight processors. Figure 3 shows performance profiles of a simulation run, where major program functions on each processor are presented in order of decreasing mean exclusive execution time. From the profiles we can see that MPI communications, including *MPI_Ssend()*, *MPI_Allreduce()*, *MPI_Barrier()*, and *MPI_Waitall()*, dominate the runtime. But the profiles provide little insight into the performance of the AMR operations or the supporting data communications with PRT.

With the performance knowledge captured for FLASH in the Hercule framework, we attempted to discover performance problems automatically. Hercule first conducts an experiment to find the first performance symptom, expensive communication cost. The output listing is shown below.

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

```
do experiment 1... ...

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In FLASH, the *AMR_Guarcell* algorithm first restricts the data at ”leaf” blocks up to the parent block, then all blocks that are leaf blocks or are parents of leaf blocks exchange guardcell data with any neighbor blocks they might have at the same refinement level. Hercule explains guardcell-filling performance from two dimensions here. It informs performance of each functional category, including *AMR_Restriction*, building tree connection, and guardcell data transportation. It also breaks down communications into collective (including *MPI_Allreduce*, *MPI_Barrier*, and *MPI_Alltoall*) and point-to-point (P2P) groups (including *MPI_Ssend*, *MPI_Irecv*, and *MPI_Waitall*). The P2Ps are used mostly in the tree-related data transfer. From figure 3 we already know that these operations dominate the runtime. Hercule discriminates the performance of these two types of communication in *AMR_Guardcell* and *AMR_Restriction*, and interprets the P2P performance as reflected in the PRT compute. The users can thereby obtain an extensive insight into the FLASH performance from the perspective of the parallel models they coded with.

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

6 Conclusions

Models of parallel computation are useful for discovering and explaining performance problems of parallel applications. For programs based on singleton models, we have shown that capturing knowledge of model behaviors, performance properties, and inference rules proves effective for diagnosis automation [8]. However, the approach will be limited in practice if we do not allow for more complex applications that combine multiple computational methods. In this paper, we extend the model-based diagnosis methodology to support compositional models that integrate singleton computational patterns. Model nesting and model restructuring are two general compositional forms for which we discuss systematic steps to generate the performance knowledge necessary for automatic diagnosis of compositional programs. Our approach addresses the performance implications of model integration so that performance losses due to model interaction can be detected and interpreted. We implemented compositional performance diagnosis in Hercule framework and tested it with two scientific applications, FLASH and PDLAHQR. The FLASH results reported here suggest that automatic diagnosis of compositional model performance is viable and effective.

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

References

[1] CLIPS: A Tool for Building Expert Systems, `http://www.ghg.net/clips/CLIPS.html`

[2] TAU Tuning and Analysis Utilities, `http://www.cs.uoregon.edu/research/paracomp/tau/tautools/`

[3] ASC/Alliances Center for Astrophysical Thermonuclear Flashes: http://flash.uchicago.edu

[4] G. Henry, D. Watkins, and J. Dongarra, A Parallel Implementation of the Nonsymmetric QR Algorithm for Distributed Memory Architectures, SIAM Journal on Scientific Computing, Vol. 24 , 1: 284 - 311, 2002

[5] P. R. Amestoy, I. S. Duff, J. Koster, and J. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications, 23:15–41, 2001.

[6] S. MacDonald, D. Szafron and J. Schaeffer, Rethinking the Pipeline as Object-Oriented States with Transformations, HIPS 2004

[7] Li Li and Allen D. Malony, Model-based Performance Diagnosis of Master-worker Parallel Computations, in the proceedings of Europar 2006.

[8] Li Li, Allen D. Malony, Knowledge Engineering for Automatic Parallel Performance Diagnosis, to appear in Concurrency and Computation: Practice and Experience.

[9] Li Li, Allen D. Malony and Kevin Huck, Model-Based Relative Performance Diagnosis of Wavefront Parallel Computations, HPCC 2006.

[10] B. Massingill and T. Mattson and B. Sanders, Some Algorithm Structure and Support Patterns for Parallel Application Programs, the 9th Pattern Languages of Programs Workshop, 2002

[11] F. Rabhi and S. Gorlatch, Patterns and Skeletons for Parallel and Distributed Computing, Springer-Verlag, 2003