# Automatic Search for Performance Problems in Parallel and Distributed Programs by Using Multi-Experiment Analysis*

Thomas Fahringer and Clovis Seragiotto, Jr.

Institute for Software Science, University of Vienna
{tf, clovis}@par.univie.ac.at

**Abstract.** We introduce Aksum, a novel system for performance analysis that helps programmers to locate and to understand performance problems in message passing, shared memory and mixed parallel programs. The user must provide the set of problem and machine sizes for which performance analysis should be conducted. The search for performance problems (properties) is user-controllable by restricting the performance analysis to specific code regions, by creating new or customizing existing property specifications and property hierarchies, by indicating the maximum search time and maximum time a single experiment may take, by providing thresholds that define whether or not a property is critical, and by indicating conditions under which the search for properties stops. Aksum automatically selects and instruments code regions for collecting raw performance data based on which performance properties are computed. Heuristics are incorporated to prune the search for performance properties. We have implemented Aksum as a portable Java-based distributed system which displays all properties detected during the search process together with the code regions that cause them. A filtering mechanism allows the examination of properties at various levels of detail. We present an experiment with a financial modeling application to demonstrate the usefulness and effectiveness of our approach.

## 1 Introduction

The performance of distributed and concurrent applications can be challenging, as an application's performance stems from several factors, including application design, software environment, and underlying hardware. Manual performance analysis is time-consuming, error-prone, and complicated. Many performance tools have been developed, which commonly produce a vast amount of performance data without interpreting performance results, relating performance information back to the input program, or guiding the user to the essential performance problems.

In previous research we have developed a Java-based specification language, JavaPSL [5], for a systematic and portable specification of large classes of experiment-related data (describe performance experiments) and performance properties (characterize a specific negative performance behavior) in distributed and concurrent programs. JavaPSL provides a high-level and portable interface to experiment-related data that enables the programmer to customize existing properties and define new ones without the need to know anything about the underlying implementation details. In addition, we developed SCALEA [9], a performance instrumentation and measurement system that computes a variety of metrics (timing and overhead information) for OpenMP, MPI, HPF, and mixed parallel/distributed programs.

In this paper we describe Aksum, a novel multi-experiment tool that tries to automate performance diagnosis of parallel and distributed programs for different problem and machine sizes. Fig. 1 gives an overview of Aksum's architecture. A *user portal* enables the user to input information about the application and the target machine on which the application will be executed. The user can select, modify and create a hierarchy of performance properties to be searched in an application, control the search time, and specify code regions of interest and conditions to stop the search process.

The user-supplied input data is provided to the *search engine*, which is in the center of Aksum and controls the entire search process. By issuing requests to the *instrumentation engine*, the search engine collects the performance information for well-selected code regions for varying user-provided problem and machine sizes. The instrumentation engine of Aksum provides an interface to SCALEA, enabling the search engine to access and traverse application files in a machine-independent way, and to modify makefiles, scripts, and the compilation command line in order to link the instrumented application with the instrumentation library provided by SCALEA. The instrumented code is submitted to the *experiment generator*.

For each instrumented application the experiment generator changes some important execution parameters that control, for instance, the set of input files and the number of threads or processes to be used for the application execution. All files for an experiment are forwarded to the *experiment launcher*, which compiles and executes the application instances on the target machine through an external scheduler. After an experiment has been executed, all performance data are transferred to the experiment data repository. The search engine evaluates the performance data and tries to determine all critical performance properties. A cycle consisting of consecutive

* This research is partially supported by the Austrian Science Fund as part of Aurora Project under SFBF1104

phases of application execution and property evaluation is continued until all experiments are done or some system or user-defined condition stops the search process. Every performance property that has been determined to be critical is dynamically displayed (together with the source code) to the user during the search process and stored in the experiment data repository. All components of Aksum can reside on any computing node that is accessible through the Internet. Although the user is given many opportunities to influence the search for performance bottlenecks, Aksum is a fully automatic tool once it is given the application sources as well as problem and machine sizes for which performance analysis should be done.

The remainder of this paper is organized as follows. Section 2 briefly describes JavaPSL. Aksum's user portal is delineated in Section 3, followed by a description of the underlying search engine in Section 4 and the experiment generator in Section 5. Results for a financial modeling application are presented in Section 6. Related work is discussed in Section 7. Finally, we conclude this paper with a summary and outline future work.

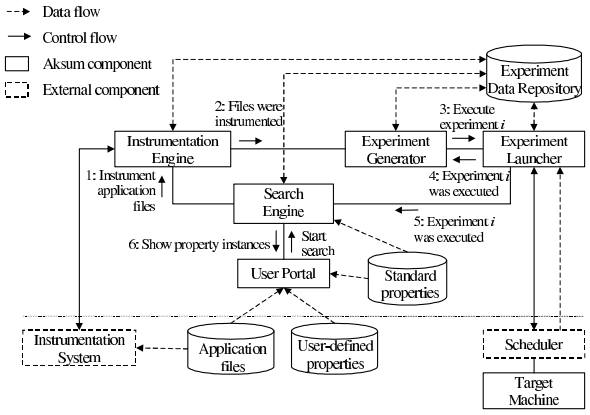


Fig. 1. Aksum's overview

## 2 JavaPSL and Performance Properties

JavaPSL [5] is a flexible API for describing experiment-related data (see Fig. 2) and performance properties of applications by using syntax and semantic rules of the Java programming language; based on it, the user can define new performance properties without the need to understand the storage format of experiment data and without changing the implementation of the search tool that makes use of this API. Even though JavaPSL can be used to describe intricate details of an experiment, this section just describes the JavaPSL features needed to understand the presented work.

In JavaPSL, an *Application* is modeled as a set of *Versions*, each of them consisting of a set of *Source Files*. An *Experiment* refers to the execution of a sequential or parallel application version with a specific set of application *Input Parameters*. *Dynamic Code Regions* describe information (e.g., execution and communication time, process/thread that executes the code region) about a specific execution of a (static) *Code Region* in a version's source code. *ParallelExperiment*, *SequentialExperiment*, *ReferenceDynamicCodeRegion,* and *ReferenceCodeRegion* are marker interfaces [7] (that is, without any method declaration) which are used to indicate semantic attributes of classes. By using these interfaces, the property definition becomes clearer, more precise, and the property evaluation potentially faster.

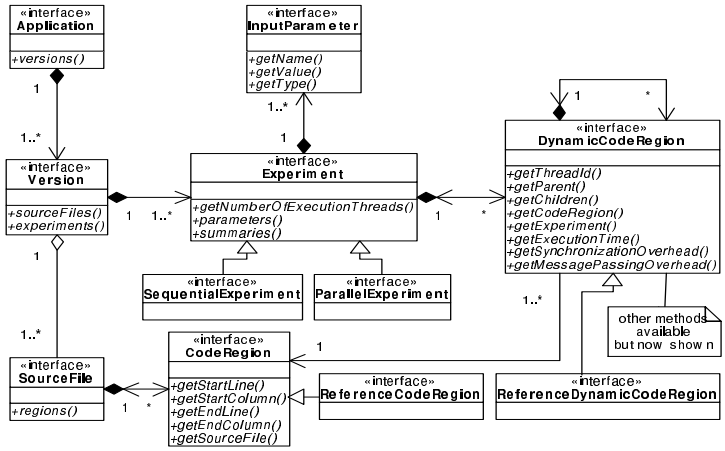


**Fig. 2.** Experiment-related data represented by JavaPSL

A performance property (e.g. load imbalance, synchronization overhead) characterizes a specific negative performance behavior of a program. Every property must implement the interface *Property*, which contains three methods:

*boolean holds( )*: returns true if the property instance holds. In this case the instance is called a true instance.

*float getSeverity( )*: returns a value between 0 and 1 indicating how severe the property is (the closer to 1, the more severe the property is)

*float getConfidence( )*: returns a value between 0 and 1 that indicates the degree of confidence in the correctness of the value returned by *holds*.

Fig. 3 shows a simple yet very common property definition example. The JavaPSL property SynchronizationOverhead is defined for a dynamic code region *cregion* and a reference to some other dynamic code region *refCregion* (e.g, the main program or the enclosing subroutine). The severity is computed as the ratio between the corresponding synchronization overhead of *cregion* and the execution time of *refCregion*. The property is true if the severity value is larger than zero.

```
public class SynchronizationOverhead
             implements Property {
  private float severity;
  public SynchronizationOverhead(
          DynamicCodeRegion cregion,
          ReferenceDynamicCodeRegion refCregion) {
    severity =(float)
               cregion.getSynchronizationOverhead()/
               refCregion.getExecutionTime();
  }
  public boolean holds( ) { return severity > 0; }
  public float getSeverity( ) { return severity; }
  public float getConfidence( ) { return 1; }
}
```

**Fig. 3.** Performance Property described in JavaPSL

In the remainder of this paper, the term *property instance* is used when referring to an instance of a performance property.

## 3 The User Portal

Aksum's user portal provides the user with a very flexible mechanism to control the search for performance bottlenecks, which is described in this section.

### 3.1. Property Hierarchy

A performance property (e.g. load imbalance, synchronization overhead) characterizes a specific negative performance behavior of a program. Every property must implement the interface *Property*, thus the term property instance refers to an instance of a class that implements this interface.

The interface *Property* contains three methods:

*boolean holds( )*: returns true if the property instance holds. In this case the instance is called a true property instance.

*float getSeverity( )*: returns a value between 0 and 1 indicating how severe the property instance is (the closer to 1, the more severe the property instance is).

*float getConfidence( )*: returns a value between 0 and 1 that indicates the degree of confidence in the correctness of the value returned by *holds*.

Properties are hierarchically organized into tree structures called *property hierarchies*, which are used to tune and prune the search for performance properties. For example, one may assume that, if an application is efficient, there is no need to compute its load imbalance. This assumption can be encoded in a specific property hierarchy by placing the property LoadImbalance under the property Inefficiency (Fig. 4). Each node in the property hierarchy represents a performance property and is described by three elements:

- *Performance property name*: the name of the performance property associated with this node; the property definition is stored in a property repository (defined by the user or provided by Aksum).
- *Threshold*: a value that is compared against the severity value of instances of the property represented by this node; if the severity value is larger than this value, the property instance is *critical* and will be included in the list of *critical* properties.
- *Reference code region*: type of the reference code region (currently main program, enclosing outermost loop or subroutine) that will be used to compute the severity of instances of the property represented by this node.

There are three standard property hierarchies provided by Aksum, covering message passing, shared memory and mixed parallel programs. The user can define and store new property hierarchies from scratch or customize predefined hierarchies. The reference code region for every property node in the predefined property hierarchies is per default set to the main program.

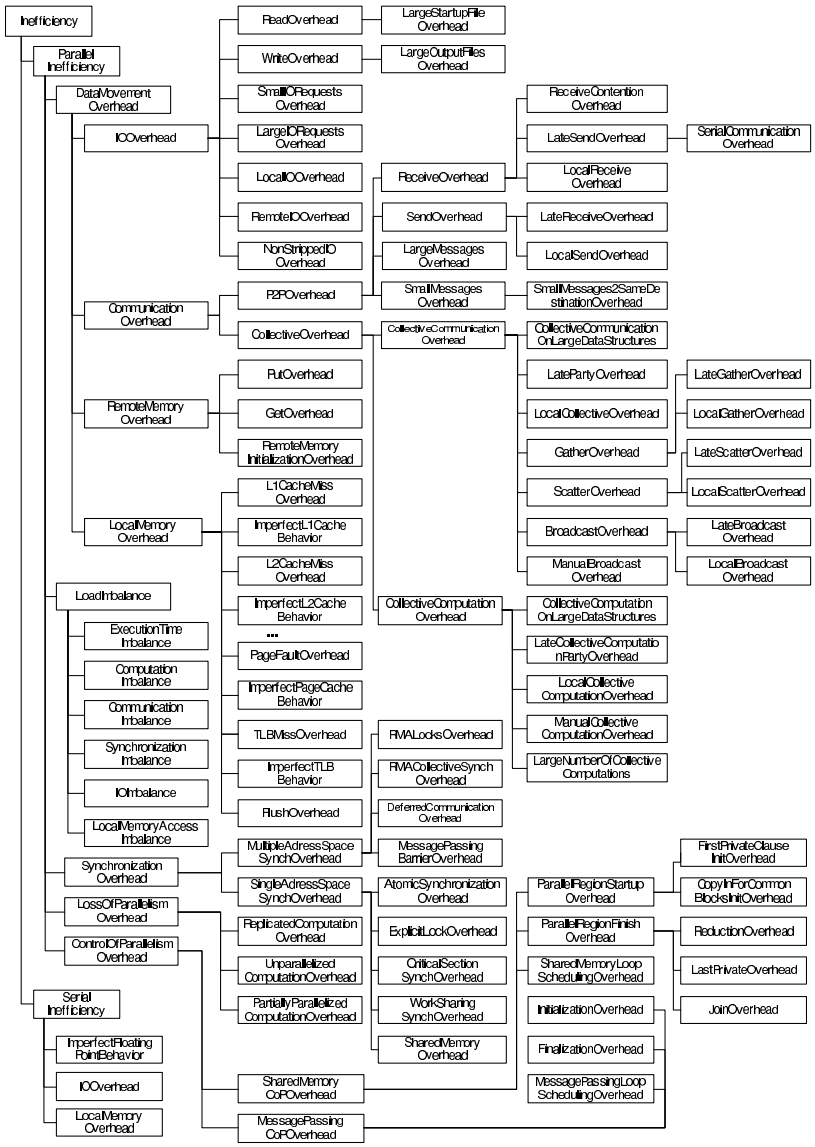


**Fig. 4.** The full Aksum property list for mixed parallel programs

### 3.2.Application Files, Command Lines and Directories

An application consists of various files – denoted *application files* in the remainder of this paper – which are divided into *instrumentable* and *non-instrumentable* files. Instrumentable files are source codes that must be instrumented for performance metrics (overheads and timing information) whereas non-instrumentable files refer to source codes the user does not want to be instrumented by Aksum and any other files necessary to execute an application (e.g., makefiles, scripts, and input files). For instrumentable files, the user can also define which code regions should be instrumented. If not specified, then Aksum assumes that the entire file must be instrumented. The user must also provide the compilation and execution command lines and associated directories for a selected application. This information is later used by the experiment generator and the experiment launcher.

### 3.3. Application Parameters

An application parameter defines a string that should be replaced in some or all of the application files and in the execution and compilation command lines before the application is compiled and executed. Application parameters are defined by the quintuplet (*name*, *search string, file set*, *value list, type*), where *name* is a unique name

that identifies the parameter, *search string* represents the string to be substituted, *file set* describes the set of application files in which the search string will be searched and replaced, *value list* denotes the list of values with which the search string will be replaced, and *type* indicates if the parameter is machine or problem size related (or neither of them). An application with a set of application files and application parameters is denoted an *application instance* if, for all application parameters, every *search string* has been replaced in the associated *file set* by one of the values in the *value list*. The generation, compilation and execution of an application instance is called an *experiment*. Fig. 5 shows user-provided values for the application parameters under the user portal. Two additional options ("Line by line" versus "Cartesian product") affect the number of experiments generated and the parameter value combinations for each experiment.

### 3.4. Timeouts and Error Recovery

The user can define the maximum number of application instance *compilations* and *executions* that can terminate abnormally (e.g. because of a system error) without canceling the search for properties. As in general it is impossible to determine whether an experiment has successfully finished or not, the current implementation of Aksum assumes that the experiment terminated normally if performance data has been stored in the experiment data repository. Aksum enables the user to select two additional options for terminating experiments:

- The execution time of an application instance exceeds a user-defined time value.
- After polling the CPU $n$ times every $k$ seconds, Aksum detects that the load of every CPU used by a given experiment is always below a certain value $f$ (where $n$, $k$ and $f$ are user-defined values). This condition represents a simple mechanism to detect a class of deadlocks during the execution of application instances.

Furthermore, the user can define a timeout limiting the overall search process time.

### 3.5. Checkpoints

In some cases, it may be useless to continue executing all application instances in the search for performance problems. For instance, if an application is efficient for the machine sizes 10, 20 and 40, one may assume that this behavior would not change for the machine sizes 15 and 30 as well. Aksum supports the user in specifying checkpoints to stop the search for properties if some condition holds. A checkpoint is defined as follows:

*op*(severity(property, code region, number of experiments)) *relop* value

where $op \in$ {maximum, minimum, average, standard deviation} and $relop \in \{>, \geq, <, \leq, =, \neq\}$. Fig. 5 shows a checkpoint definition that stops the search process if the average inefficiency for the entire program in the last 5 experiments is above 0.75 with standard deviation less than or equal to 0.1.

## 4 The Search Engine

The Search Engine controls the search for performance properties based on the experiments conducted. The Search Engine automatically

- decides which code regions should be instrumented and the performance overheads and timings that should be determined for these regions, and
- controls the evaluation of properties based on the property hierarchy.

Currently, the search engine issues requests for instrumenting all subroutines, outermost loops, subroutine calls (in particular, calls to the MPI library), and OpenMP constructs within the file regions of interest. After the instrumentation (done by SCALEA), an application can be accessed by the experiment generator and the experiment launcher in order to have its parameters replaced (Section 3.3) and to be compiled and executed on the target machine. Every time an experiment terminates normally, the resulting performance data is stored in the experiment data repository and the search engine is notified about that. Thereafter, the search engine evaluates the property hierarchy as follows:

- Every property in the property hierarchy is tried to be instantiated with the data (represented in JavaPSL) stored in the experiment data repository.
- All true property instances are examined against the threshold for the given property in the property hierarchy. If the severity value of the property instance is above the threshold, then the instance is included in the list of *critical property instances,* which is the main output of the search engine.

### 4.1. Heuristics for Pruning the Evaluation of Property Hierarchies

The search engine traverses the property hierarchy in pre-order; it prunes the evaluation of the property hierarchy and avoids the creation of false and non-critical property instances by using a well-defined heuristic. Assume that an instance of property $\delta$ is true for a code region $q$ in an experiment iff an instance of property $\delta$' is also true for $q$ in the same experiment. Then, we do not need to test whether $\delta$ holds for a certain code region in an experiment if we already know that $\delta$' does not hold for the same code region in this experiment.

Before we define the pruning heuristics of Aksum we describe some important notation. A *static code region* (or simply code region) is a non-empty set of program statements with multiple control flow entry and exit

points. If a code region $q$ is executed by a thread $t$ in a process $p$, we call a *child* of $q$ in that experiment any code region executed by thread $t$ in process $p$ after the execution of $q$ has started but not yet ended. Any code region that has $q$ as a child is a *parent* of $q$ in that experiment. Moreover, if $q$' is a child of $q$ in a certain experiment, and there is no code region $q$'' that is a parent of $q$' and child of $q$, then $q$' is an *immediate child* of $q$, and $q$ is the *immediate parent* of $q$' in that experiment.

Now, let $q$ be a code region, $e$ an experiment, and DC($q$, $e$) = { ($p$, $t$, $q$, $y$) | $t$ is a thread in process $p$ that executes $q$ in experiment $e$, and $y$ is the immediate parent of $q$ in experiment $e$}. Every element $d$ = ($p$, $t$, $q$, $y$) in DC($q$, $e$) represents a particular execution of a code region and is called a *dynamic code region*. We will denote $y$ as *parent*($d$), $e$ as *experiment*($d$), and $q$ as *codeRegion*($d$).

Next, we define a *containment relation* between (dynamic) code regions:

- Given two dynamic code regions, $d$ and $d$', then $d$ contains $d$' if $d = d$' or there is a sequence of dynamic code regions ($d_1$,…, $d_n$) such that $d_1 = d$, $d_n = d$', and $d_i$ = parent($d_{i+1}$) $\forall$ $i$, $1 \le i <$ n.
- Given a dynamic code region $d$ and a static code region $q$, then $d$ contains $q$ if codeRegion($d$) = $q$ or there is a sequence of dynamic code regions ($d_1$,…, $d_n$) such that $d_1 = d$, codeRegion($d_n$) = $q$, and $d_i$ = parent($d_{i+1}$) $\forall$ $i$, $1 \le i < n$.
- Given a dynamic code region $d$, a static code region $q$, and an experiment $e$, then $q$ contains $d$ in experiment $e$ if there is a dynamic code region $d$' such that $d$' contains $d$, experiment($d$') = $e$, and codeRegion($d$') = $q$.

For every node $k$ in the property hierarchy, the search engine loads the definition of the associated property $p$ from a property repository and analyzes its constructor and all property instances that have been created for all properties along the path from the property tree root to $k$ in order to determine the actual arguments that must be used to create instances of $p$. Based on the definitions above, we can now formally define the heuristics used by the search engine of Aksum:

Let $\delta_k$ and $\varphi_k$ respectively denote the property and threshold of a node $k$ in the property tree. For simplicity, we assume that $\delta_k$ has only one constructor. Now, assume that the search engine is just considering $k$ for its search for performance properties and ($a_1$, …, $a_n$) is a list of actual arguments compatible with the formal constructor parameters of $\delta_k$. The search engine tries to create an instance of $\delta_k$ using ($a_1$, …, $a_n$) as actual arguments if either $k$ has no predecessor node in the property hierarchy, or if there exists a predecessor node $g$ of $k$ and an instance $r$ of $\delta_g$ generated with the actual arguments ($b_1$, …, $b_m$) such that *r.holds()* is true, *r.getSeverity()* $\ge \varphi_g$, and any of the following conditions is true:

1. $\exists$ $i$, $j$, $1 \le i \le n$, $1 \le j \le m$: $a_i$ and $b_j$ are dynamic code and $b_j$ contains $a_i$.

   A property can be instantiated with a specific dynamic code region if an instance of a predecessor property in the property hierarchy is critical for either the same dynamic code region or a parent of this dynamic code region.
2. $\exists$ $i$, $j$, $v$, $1 \le i \le n$, $1 \le v < j \le m$: $a_i$ is a dynamic code region, $b_j$ is a static code region, $b_v$ = experiment($a_i$), and $b_j$ contains $a_i$ in experiment $b_v$.

   A property can be instantiated with a specific dynamic code region $a_i$ if, in the experiment associated to $a_i$, an instance of a predecessor property in the hierarchy is critical for the static code region that $a_i$ (or a parent of $a_i$) refers to.
3. $\exists$ $i$, $u$, $j$, $1 \le u < i \le n$, $1 \le j \le m$, $a_i$ is a code region, $a_u$ an experiment, $b_i$ a dynamic code region, $a_u$ = experiment($b_j$), and $b_j$ contains $a_i$.

   A property can be instantiated for a specific static code region and a specific experiment if an instance of any predecessor property in the hierarchy is critical for a dynamic code region referring to this code region and generated by this experiment.
4. $\exists$ $i$, $u$, $j$, $v$, $d$: $1 \le u < i \le n$, $1 \le v < j \le m$: $a_i$ and $b_j$ are code regions, $a_u$ and $b_v$ refer to the same experiment, and $d$ is a dynamic code region such that codeRegion($d$) = $a_i$, experiment($d$) = $a_u$, and $b_j$ contains $d$ in $a_u$.

   A property can be instantiated for a specific static code region and a specific experiment if an instance of any predecessor property is critical for the same static code region and the same experiment.
5. $\exists$ $i$, $u$, $j$, $d$: $1 \le u < i \le n$, $1 \le j \le m$: type($a_i$) = CodeRegion, type($a_u$) = Experiment, type($b_j$) = CodeRegion, type($d$) = DynamicCodeRegion, codeRegion($d$) = $a_i$, experiment($d$) = $a_u$, $a_i$ contains $d$ for $a_u$, and $\forall$ $v$, $1 \le v < j$, type($b_v$) $\ne$ Experiment.

   A property can be instantiated for a specific static code region and a specific experiment if an instance of any predecessor property in the hierarchy is critical for the same static code region independently of the experiment.

## 5 The Experiment Generator

The experiment generator creates application instances and submits them to the experiment launcher. Based on the description of a generic (application) input parameter $v$ (see Section 3.3), a quintuple (*name*($v$), *searchString*($v$), *fileSet*($v$), *valueList*($v$), *type*($v$)), we define an application instance as follows[1]:

[1] Value lists may contain undefined (i.e., non-specified) strings.

Let ($v_1$,…, $v_n$) be the list of application input parameters. The set of instrumented application files is denoted an *application instance* AppInst($s^{v_1}$,..., $s^{v_n}$) iff $\forall i$, $1 \le i \le n$, the string *searchString*($v_i$) has been substituted by a defined string $s^{v_i}$ of *valueList*($v_i$) in every file of *fileSet*($v_i$).

Experiments are generated according to the following policies:

- Let ($v_1$,…, $v_n$) be the list of input parameters, and $s^{v_1}$,..., $s^{v_n}$elements in *valueList*($v_1$), …, *valueList*($v_n$), respectively. Recalling (Section 3.3) that the user has two options to control the generation of application instances, "cartesian product" or "line by line", the experiment generator creates AppInst($s^{v_1}$,..., $s^{v_n}$) for every different combination of (specified) values in the value lists if the cartesian product option was chosen, while, for the option line by line, the experiment generator creates AppInst($s^{v_1}$,..., $s^{v_n}$) iff position($s^{v_i}$) = position($s^{v_j}$) $\forall$ $i, j$ : $1 \le i, j \le n$, where position($s^{v_i}$) denotes the position of an element $s^{v_i}$ in *valueList*($v_i$).
- Let $w_1$ = AppInst($s^{v_1}$,..., $s^{v_n}$) and $w_2$ = AppInst($\bar{s}^{v_1}$,..., $\bar{s}^{v_n}$). Then $w_1$ is generated (and later on also executed by the experiment launcher) before $w_2$ iff $\exists$ $k$ such that $1 \le k \le n$ and position($s^{v_i}$) $\le$ position($\bar{s}^{v_i}$) $\forall$ $i$, $1 \le i \le k$. This option enables the specification of an order for the generation of application instances, which can be important when defining checkpoints.
- A value like {mango, passion fruit, banana, pineapple} has a special meaning in a value list. Let ($v_1$,…, $v_n$) be the list of input parameters, *valueList*($v_j$) = ($s^1_j$, …, $s^m_j$), $1 \le j \le n$, and $\phi_1$, …, $\phi_p$ arbitrary strings (for simplicity, suppose that no commas or braces occur inside these strings); let also *undef* represent the undefined value, and $\phi$ the concatenation of the character ´{´, the ings $\phi_1$, …, $\phi_p$, and the character ´}´, with a character ´,´ between each string $\phi_i$. Finally, let s($\phi$, $i$), $1 \le i \le p$, denote the string where each occurrence of $\phi$ was replaced with $\phi_i$. If $\exists$ k, $1 \le k \le m$: $s^k_j$ contains $\phi$, then the application instances are generated **as if**, $\forall$ $j$, $1 \le j \le n$, *valueList*($v_j$) were:
  - if $j \ne i$ and the option "line by line" is selected: ($s^1_j$, …, $s^{k-1}_j$, $\underbrace{s^k_j, \quad s^k_j, \quad \ldots, \quad s^k_j}_{p \text{ times}}$, $s^{k+1}_j$, …, $s^m_j$)
  - if $j \ne i$ and the option "cartesian product" is selected: ($s^1_j$, …, $s^{k-1}_j$, $s^k_j$, $\underbrace{undef, \quad \ldots, \quad undef}_{(p-1) \text{ times}}$, $s^{k+1}_j$, …, $s^m_j$)
  - if $j = i$: ($s^1_j$, …, $s^{k-1}_j$, s($\phi$; 1), s($\phi$, 2), …, s($\phi$, $p$), $s^{k+1}_j$, …, $s^m_j$)

Suppose, for instance, that *valueList*($v_1$) = (X, Y, Y) and *valueList*($v_2$) = (x, y{1, 2, 3}, z). With the option "line by line", they are equivalent to (X, Y, Y, Y, Z) and (x, y1, y2, y3, c), while, with the option "cartesian product", to (X, Y, undef, undef, Z) and (x, y1, y2, y3, z).

The string {a:b:c}, where a, b and c are real numbers, is a special shortcut for {$a_0$,…, $a_{z-1}$}, where $z = \lfloor (b - a) \div c \rfloor$ and $\forall$ $j$, $1 \le j \le z$, $a_j = a + j * c$.

## 6 Experimental Results

In this section, we present a performance analysis for a financial modeling application conducted on an SMP cluster with 16 quad nodes connected through Fast Ethernet. The backward pricing application implements the backward induction algorithm to compute the price of an interest rate dependent product [2]. The backward induction algorithm has been implemented as an HPF+ [1] code based on which the VFC compiler [1] generates a mixed OpenMP/MPI code. Aksum analyses the performance of this code at the OpenMP/MPI program level.

Fig. 5 shows the user portal of Aksum, in which the user pre-selects and modifies the property hierarchy, the application input parameters and their value sets, the execution command line, and the execution directory. For instance, *P* is a machine size parameter that indicates the number of processors (covering 1, 2, 4, 8, and 16 processors) to be considered by Aksum.

Based on the user provided input data, the search engine of Aksum automatically determines that seven performance properties in the property hierarchy are critical for this code (see Fig. 6), where the properties are presented in ascending order of severity. Initially, the user portal displays only the property names for those instances whose severity is above the user-defined threshold. The property instances can be shown by expanding each property name. For every instance the corresponding code region and severity value are indicated. Instrumentation directives automatically inserted by Aksum are also shown. In the backward pricing application, the most serious performance property is ExecutionTimeLoadImbalance, which has an instance that holds for the main (entire) program with severity value 0.80 (see the entry *BW_HALO_3 0.80*). The same property holds for the sub-region of the main program indicated by the entry *BW_HALO_2 0.80*.

The severity of the ExecutionTimeLoadImbalance property instances for the entire application increases with the number of execution threads (not shown in Fig. 6), from 0.01 for 2 CPUs to 0.80 for 64 CPUs. This behavior also explains the increasing severity values for the Inefficiency property (varying from 0.05 for 2 CPUs to 0.79 for 64 CPUs – see the Inefficiency diagram in the upper right window of Fig. 6). All other properties in the prop-

erty hierarchy have lower severity values (SynchronizationOverhead: 0.01, MessagePassingOverhead: 0.17 with 64 CPUs, for the other machine sizes 0.00).

The main program calls the subroutine BW, which calls subroutine COMPUTE_SLICE. As the property Inefficiency is not critical for COMPUTE_SLICE, and since the critical instances of these properties have always approximately the same value for both the main program and the subroutine BW, we conclude that performance tuning should mainly concentrate on subroutine BW.

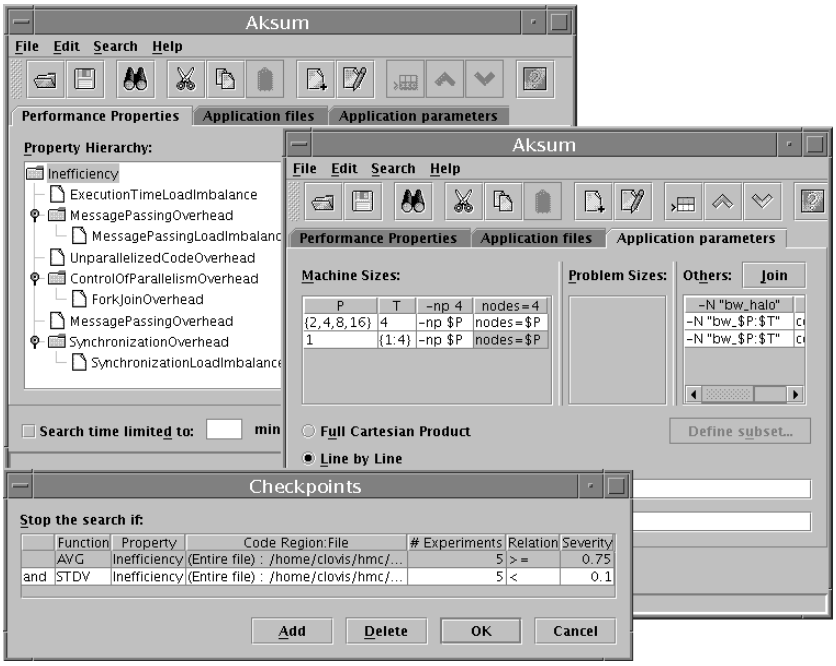


**Fig. 5.** User-provided input data under Aksum's user portal

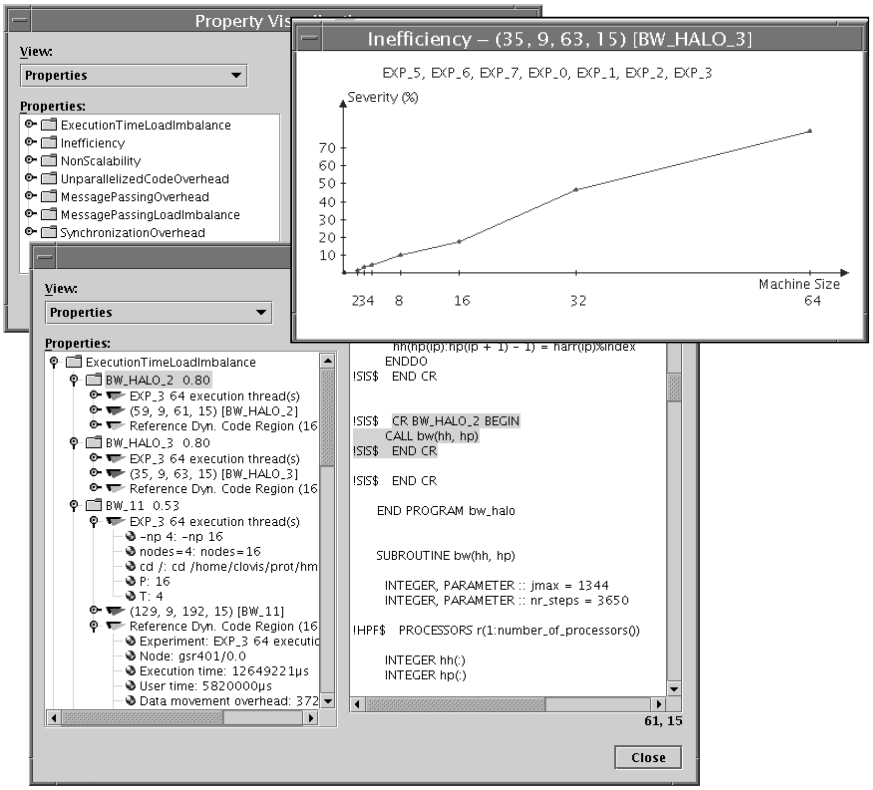


**Fig. 6.** Snapshots of Aksum's property visualization for the Backward Pricing application

## 7 Related Work

Various groups have developed performance tools that are oriented towards automatic analysis. Paradyn [8] performs an automatic online analysis by searching for performance bottlenecks based on thresholds and a predefined (but immutable) set of hypotheses.

The European working group APART [4] defined a specification language for performance properties of parallel programs which has initiated the development of JavaPSL and Aksum.

Kappa-Pi [3] and Earl/Expert [11] are post-mortem tools that search for performance properties in message passing trace files in combination with source code analysis. Expert also covers OpenMP and mixed parallel programs, and uses the concept of performance properties organized in a hierarchy. Performance properties are also used in the Peridot project [6].

In [10] an approach is described that uses machine learning to detect performance problems in message passing codes. A decision tree trained for different target architectures is employed to detect individual communication performance problems.

All of the previously mentioned tools concentrate performance analysis on single experiments only whereas Aksum examines the performance for multi-experiments based on an arbitrary user-provided set of machine and problem sizes. Aksum allows the user to modify and add new properties and property hierarchies. Aksum also provides a rich set of pre-defined properties and several possibilities to impact the search for properties. We believe that Aksum is also more flexible to customize the automatic search for parallel and distributed programs compared to other work

## 8 Conclusion

We have presented Aksum, which is a highly flexible multi-experiment automated performance analysis tool for distributed and concurrent architectures. Based on a user-provided set of problem and machine size values, Aksum tries to automatically detect all critical performance problems and to guide programmers to performance problems that should be addressed for performance tuning. Aksum provides a very flexible mechanism to create and customize performance properties of interest and to control the search process. Although the user is given many opportunities to influence the search for performance bottlenecks, Aksum is a fully automatic tool once it is given the application sources as well as problem and machine sizes for which performance analysis should be done.

We are currently extending Aksum to enable the specification of search algorithms. Moreover, we plan to extend Aksum by online performance analysis based on dynamic instrumentation.

## References

1. Siegfried Benkner. VFC: The Vienna Fortran Compiler. *Journal of Scientific Programming*, 7(1): 67-81, 1999.
2. Engelbert Dockner and Hans Moritsch. Pricing Constant Maturity Floaters with Embedded Options Using Monte Carlo Simulation. Aurora Technical Reports, University of Vienna. January, 1999.
3. Antonio Espinosa, Tomàs Margalef, and Emilio Luque. Automatic Performance Evaluation of Parallel Programs. IEEE Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing. January, 1998.
4. Thomas Fahringer, Michael Gerndt, Bernd Mohr, Feliz Wolf, Graham Riley, and Jesper Larsson Träff. Knowledge Specification for Automatic Performance Analysis. http://www.fz-juelich.de/apart-1/reports/wp2-asl.ps.gz. January, 2001.
5. Thomas Fahringer and Clovis Seragiotto. Modeling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. *Proceedings SC 2001*, November, 2001.
6. Michael Gerndt, Andreas Schmidt, Martin Schulz, Roland Wismüller. Performance Analysis for Teraflop Computers – A Distributed Approach. 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing. January, 2002.
7. Mark Grand. *Patterns in Java, Volume 1*. Wiley, 1998.
8. Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, November 1995.
9. Hong-Linh Truong, Thomas Fahringer. SCALEA: A Performance Analysis Tool for Distributed and Parallel Program. *8th International Europar Conference*, August, 2002.
10. Jeffrey Vetter. Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficiencies. *In Proceedings of the 14th International Conference on Supercomputing*, pp. 245-254, Santa Fe, New Mexico, May, 2000.
11. Felix Wolf, Bernd Mohr. Automatic Performance Analysis of SMP Cluster Applications. Internal Report, Forschungszentrum Jülich GmbH. August, 2001.