Chapter 1
PERISCOPE: An Online-Based Distributed Performance Analysis Tool

Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt

Abstract This paper presents PERISCOPE - an online distributed performance analysis tool that searches for a wide range of performance bottlenecks in parallel applications. It consists of a set of agents that capture and analyze application and hardware-related properties in an autonomous fashion. The paper focuses on the Periscope design, the different search methodologies, and the steps involved to do an online performance analysis. A new graphical user-friendly interface based on Eclipse is introduced. Through the use of this new easy-to-use graphical interface, remote execution, selection of the type of analysis, and the inspection of the found properties can be performed in an intuitive and easy way. In addition, a real-world application, namely, the GENE code, a grand challenge problem of plasma physics is analyzed using Periscope. The results are illustrated in terms of found properties and scalability issues.

1.1 Introduction

Performance is one of the key concerns for the application developers and the computational resource providers of the scientific community. The application developers endeavor writing efficient large-scale scientific codes that make optimal use of supercomputers or any other computational resource. Very often, they fail owing to the required knowledge about the architectures, memory hierarchy, and networks connecting the processors. The factors which determine a program's performance, such as, memory usage, I/O, compilers, operating system and so forth, are complex, inter-related, and frequently hidden from the programmer.

Performance analysis tools help users in writing efficient codes for current High Performance Computing (HPC) machines. These tools can provide the user with

Shajulin Benedict, Ventsislav Petkov and Michael Gerndt
Technische Universität München
Fakultät für Informatik I10, Boltzmannstr. 3, 85748 Garching, Germany

measurements of the program's performance and locate various bottlenecks. Bottlenecks are places in the execution path where execution time is lost due to inefficient resource usage. Based on the identified bottlenecks, users can do modifications to improve the application's runtime behavior. Since measuring performance data and storing those data for further analysis is often done in a not very scalable approach, most tools are limited to experiments with a small number of processors.

The traditional way of conducting performance analysis and tuning for high performance computing has been an off-line search approach requiring strong involvement of the user. This search has a potential problem with large performance datasets and long analysis times for large-scale scientific applications. It remains a challenge for application developers to analyze the bottlenecks of their applications when scaling to larger parallel machines. To investigate the runtime behavior of large experiments, performance analysis has to be done online in a distributed fashion, eliminating the need to transport huge amounts of performance data through the parallel machine's network and to store those data in files for further analysis.

An online-based performance analysis system using expert knowledge for identifying bottlenecks in the applications, in general, follows four steps for capturing performance properties (Fig. 1.1). As shown, the application is instrumented based

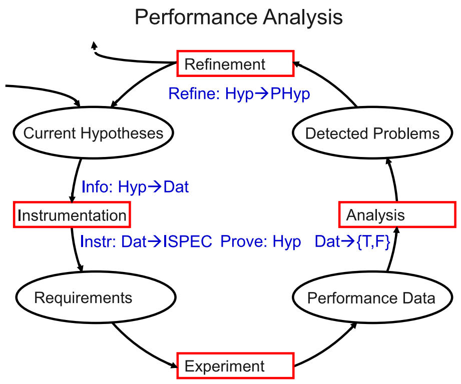


Fig. 1.1: Cyclic representation of performance analysis

on the initial hypotheses of potential performance properties. During an experiment executing the application on the parallel system, appropriate performance data are collected. These data are then inspected to prove which of the hypotheses hold. In the refinement step, the found properties might be refined to identify more specific performance problems. All four steps are executed in a cyclic fashion until no more precise properties can be found. This cyclic approach can of course be automated and executed in an online fashion.

Although, there are numerous performance analysis tools on the market, they face challenges in usabiliy, scalability, and single node performance analysis. Periscope [5] is a distributed online performance analysis tool currently under development at Technische Universität München that addresses the above mentioned challenges. It consists of a set of autonomous agents that search for performance properties. Each agent is responsible for analyzing a subset of the application's processes and threads. The agents request measurements from the monitoring system, retrieve the data, and use it to identify performance properties.

Periscope is currently extended within the German project ISAR (Integrated System and Application Analysis for Massively Parallel Computers in the Petascale Range[1]) funded until 2011. The main goal of the ISAR project is the realization of an integrated scalable system that can be used in production environments.

This paper gives a short overview of Periscope's design. It also presents for the first time the usage model of Periscope which is based on the recent work within the ISAR project. It introduces the new graphical user interface which facilitates the user's interaction with Periscope. In addition, we give some results from a large-scale run of a real-world application, namely, the Gyrokinetic Electromagnetic Numerical Experimental code (GENE) from the Max Planck Institute for Plasma Physics in Garching.

All the experiments were done on our Altix 4700 at Leibniz Rechenzentrum (LRZ) in Garching. The Altix supercomputer consists of 19 NUMA-link4 interconnected shared memory partitions with over 9600 Itanium 2 cores with an aggregated peak performance of over 60 TFlops.

The paper is organized as follows: Section 1.2 introduces related work and Section 1.3 describes the design of Periscope with an detailed overview of the analysis agent's data capturing mechanism. The usage model is presented in Section 1.4 and the distributed performance analysis is illustrated in Section 1.5.

1.2 Related Work

Just as there are several different classes of parallel hardware and parallel programming languages, so too are there several distinct types of performance analysis tools. Each of them has a number of concreate realizations. The most notable ones are Paradyn, TAU, Vampir, KOJAK, SCALASCA, and mpiP.

Paradyn [9, 10], developed during 1990s, automates performance analysis. The toolkit uses a performance consultant that does dynamic instrumentation and searches for bottlenecks based on summary information during the program's execution. The tool has the capability to scale to large numbers of processors.

TAU [12] does offline analysis of performance issues in the application based on trace-based and profiling-based approaches. It provides a wide variety of graphical and text-based displays. In addition, it provides the PerfExplorer which uses statistical analysis to detect performance problems and allows to compare multiple application runs.

[1] http://www.in.tum.de/en/forschung/verbundprojekte/clusteraktivitaeten/isar.html

Vampir [1] provides a trace-based performance analysis framework that converts performance data obtained from a program into different performance views and supports navigation and zooming within these displays. It implements event analysis algorithms and customizable displays which enable interactive rendering of the collected monitoring data. Recently, Vampir's scalability was extended via a parallel analysis server. But still, huge trace files are generated and have to be manually analyzed afterwards.

KOJAK [13] does trace-based analysis of parallel applications. It includes Expert, a component that automatically deduces performance properties from the trace files and provides a user-interface for investigating the found types of properties, the processes where a property occurs, as well as its function or OpenMP region.

SCALASCA [4] (SCalable performance Analysis of LArge SCale Applications) is a scalable trace analysis tool based on KOJAK. It extended KOJAK's approach for automatically searching performance properties by exploiting both distributed memory and parallel processing capabilities. Instead of sequentially analyzing a single global trace file, it uses a technique based on execution replay to gather performance data for detecting formalized properties.

mpiP [7] tool is more oriented towards identifying performance problems for MPI-based applications. It collects the performance data statistically. All information is sampled locally and not exchanged between processes during runtime. The results of the experiments are only merged after program termination as a post-mortem approach.

Our approach advances the state of the art in the following aspects:

1. Performs a distributed search of properties.
2. The performance data are processed while the application runs.
3. Periscope knows and exploits the structure of the application. This information is generated by source code instrumentation and is used by the analysis agents.
4. It does not use trace-based analysis and hence avoids large datasets.
5. It has a user-friendly interface that supports remote execution on large and distributed machines and inspection of the application's performance properties.

1.3 Periscope Design

Periscope's design provides mechanisms to search for performance properties in a distributed fashion based on a master agent for management purposes, analysis agents for finding problems on individual processors, and a few other agents responsible for communication. It is devised such that the users can investigate performance problems on large-scale runs delving into the single node especially memory-related performance bottlenecks and MPI/OpenMP issues.

It can be used in interactive application runs as well as in batch runs. The analysis is executed in an iterative fashion, i.e., focusing on more specific and precise properties in multiple executions of a program phase. The program phase is either a special code region marked by the user or the whole code. While in the first case, the execution is suspended when the end of the phase region is reached and the processes are released afterwards for a next execution, in the latter case the application is automatically restarted for another execution of the phase.

In this section, we discuss Periscope's architecture, the agent's data capturing mechanism, different types of search strategies and the Eclipse plugin [3] implementing the user interface.

1.3.1 Periscope Architecture

The Periscope architecture consists of four major entities as shown in Fig. 1.2 that can perform distributed and online-based performance analysis of any scientific application.

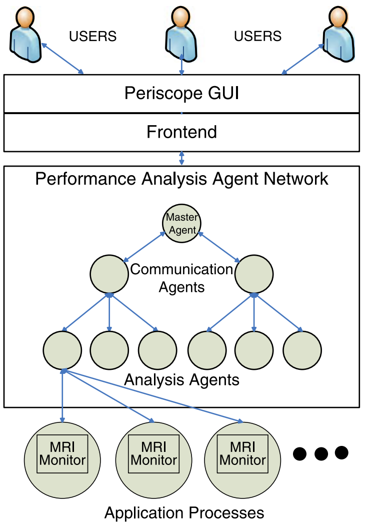


Fig. 1.2: Periscope Architecture

1. The *User-Interface* is a convenient and feature-rich entity that displays the results of the runtime analysis by directly mapping the detected properties to the source code. This entity benefits Periscope users by providing many advanced features like code indexing, refactoring, syntax highlighting, and so forth.
2. The *Frontend* starts the application and analysis agents based on the specifications provided by the user, namely, number of processes and threads and so on. It is responsible for restarting the application whenever the agent network did not complete the search before the application terminated.
3. The *Analysis agent network* consists of three different agents, namely, master agent, communication agent and analysis agent. The *master agent* forwards commands from the frontend to the analysis agents and receives the found performance properties from the individual analysis agents and forwards them to the frontend. The *communication agents* combine similar properties found in their

sibling agents and forward only the combined properties. The *analysis agents* are responsible for performing the automated search for performance properties. During the search they access the monitor linked to the application processes via the Monitoring Request Interface (MRI).

4. The *MRI monitors* provide an application control interface. They deal more with hardware and software sensors to measure the performance data.

1.3.2 Agent's Data Capture Mechanism

The agents play a vital role in the search for performance properties in the processes or threads of the application. The agent design consists of two main parts, namely, the agent and the monitor. The main components of the agents are the agent control, the search strategy, and the experiment control. Figure 1.3 presents the agent design and the sequence of operations involved in capturing performance data. In general,

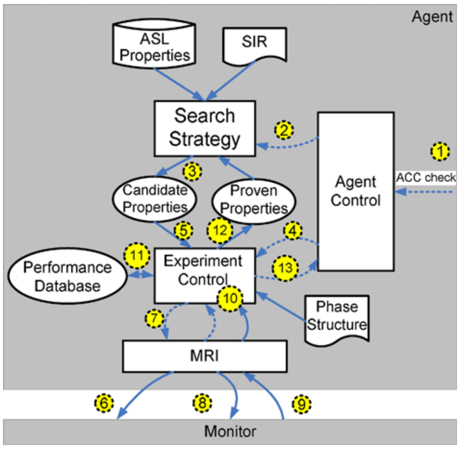


Fig. 1.3: Sequence of operations performed by agents to capture performance data

any scientific application would have computational intensive iterations. The user can mark those regions as user-region or phase-region, so that, it can be used to perform a multistep search. In order to perform such search autonomously, agents are involved in the following phases of the performance data capturing mechanism in Periscope:

1. initialization phase
2. execution phase
3. data collection phase
4. evaluation phase

During the *initialization phase* (1 to 3 in Fig. 1.3), the master agent starts the performance analysis search which triggers the search strategy and initializes the candidate set and triggers a new experiment. The master agent starts the performance analysis via the Agent Control and Command (ACC) message `CHECK`. Before the message is sent, the application is started and suspended in the initialization of the monitoring library linked to the application. In addition, the analysis agent is instructed to attach to the application via the monitor. The initial candidate set is determined by a search strategy based on the set of properties found in the previous step. At the beginning, the set of evaluated properties is empty.

The *execution phase* (4 to 8 in Fig. 1.3) of performance data capture mechanism involves triggering a new experiment, checking for missing performance data, sending MRI requests, releasing the application, and configuring the monitors. After the candidate set is determined, the agent control starts a new experiment. The experiment control accesses all the properties in the candidate set and checks whether the required performance data for proving the property are available. If not, it configures the monitor via new MRI measurement requests. The source code instrumenter used for the insertion of monitoring library calls generates information about the program's regions (main routine, subroutines, loops, etc.) in the Standard Intermediate Program Representation (SIR) format developed by the European American APART working group on automatic performance analysis tools. The requests, such as, measure the number of cache misses in the parallel loop on line 52 in file foo.f, are sent using sockets to the monitoring library. Once all the properties were checked for missing performance data, the experiment is started. The MRI provides an application control interface that allows the experiment control to release the suspended application and to specify when the application is to be suspended again to retrieve the performance data. During program execution, the monitoring library checks whether the end of the current phase is reached and configures hardware and software sensors for measuring the requested performance data.

In the *data collection phase* (9 to 11 in Fig. 1.3), a specific mechanism is used to collect performance data. In this lieu, data are inserted into a summary table. When the application is suspended again the experiment control is informed by the MRI and it retrieves the measured performance data via the MRI into the internal performance database.

Finally, during the *evaluation phase* (12 in Fig. 1.3), the candidate properties are evaluated and the sequence of operations mentioned above are repeated. Evaluation of the candidate performance properties is done by the experiment control and they are inserted into the proven properties set. At the end of this search step, the control is returned to the agent control and the cycle of the analysis steps is repeated.

1.3.3 Search Strategies

Currently, three search strategies are supported: one for analyzing the single-node performance, another for investigating the MPI communication behavior and one combining both types. Characteristic for the first search strategy is that it is a multi-step one while the MPI is a single-step operation.

1.3.3.1 Single-Node performance

There are two strategies for analyzing the code's efficiency based on stall cycle counters. They both use the PAPI library which provides a portable access to hardware performance counters. These strategies give an insight into the pipeline execution and the usage of the cache memory. The first implemented search strategy is the so called *StallCycleAnalysis*. It analyzes the defined phase region for stall cycles and, depending on their number, it can either refine the property hierarchy or stop the analysis. After detecting and proving all performance problems in the current region this strategy continues with the nested code parts and function calls. The newly created hypotheses are based on the detected properties in the parent regions in order to reduce the measurement overhead.

The second available strategy is *StallCycleAnalysisBreadthFirst*. Its fundamental idea is quite similar to the previously discussed stall cycle analysis. The main difference is that this one detects the bottlenecks for all regions in one step. It too requires multiple executions of the phase region to gather all the performance data since only a limited number of hardware counters is available.

1.3.3.2 MPI Communication Behavior

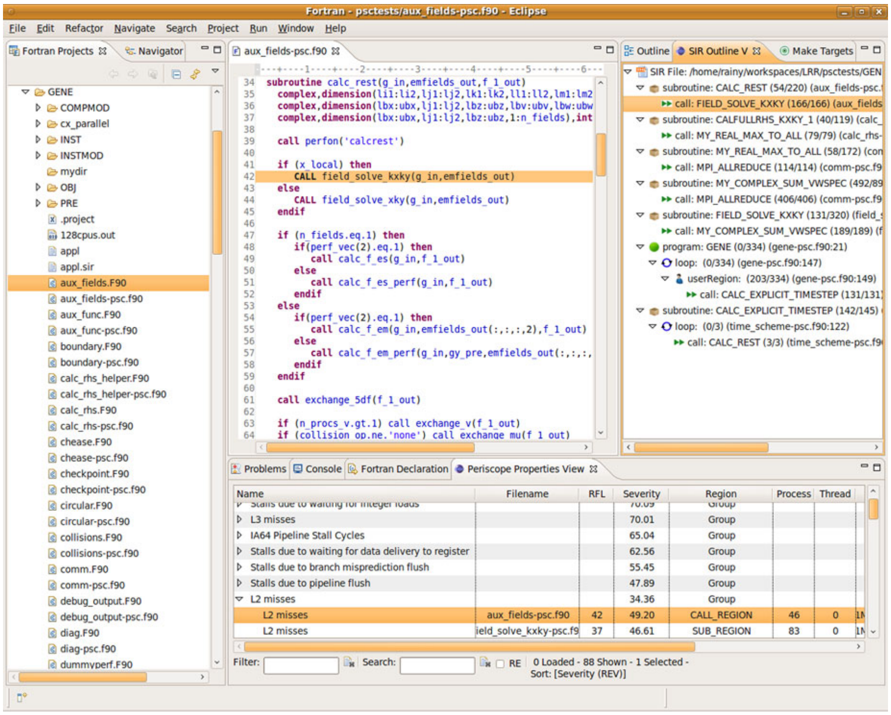
Improving the communication pattern and finding performance imbalance is a crucial optimization step for MPI applications. In order to help the developer in tuning his program, Periscope implements a MPI search strategy. It is able to detect different synchronization problems, such as, late sender and receiver. In contrast to the other strategies, this one gathers all the necessary runtime information in only one run of the phase region. The main reason for this is that the properties have no special hierarchy, thus no further refinement is required.

1.3.3.3 Aggregated Strategy

In addition, a meta strategy called *AllStrategy* exists. It does neither define any initial hypotheses nor provide any refinement algorithms. Its purpose is to combine the strategies for measuring both the single-node performance and the MPI behavior. In the beginning the former group is activated. When the stall cycle analysis is completed, the MPI one is started. The result is a complex profile that gives a deeper insight into the runtime performance of MPI programs.

1.3.4 User Interface

Periscope provides a convenient GUI as shown in Fig. 1.4 aiming at enhancing the analysis and post-processing of the found performance properties. It is developed as a plug-in for Eclipse in order to integrate it with other available programming tools such as the C/C++ Development Environment (CDT), Fortran IDE (Photran), Remote System Explorer (RSE), etc. It currently consists of three interconnected views that present the detected properties and also provide an overview of the instrumented code regions. Due to the textual character of the bottlenecks stored by


Fortran - psctests/aux_fields-psc.f90 - Eclipse
File Edit Refactor Navigate Search Project Run Window Help
Fortran Projects
Navigator
aux_fields-psc.f90
Outline
SIR Outline V
Make Targets
Problems
Console
Fortran Declaration
Periscope Properties View

```
GENE
  COMPMOD
  cx_parallel
  INST
  INSTMOD
  mydir
  OBJ
  PRE
  .project
  128cpus.out
  appl
  appl.sir
  aux_fields.F90
  aux_fields-psc.f90
  aux_func.F90
  aux_func-psc.f90
  boundary.F90
  boundary-psc.f90
  calc_rhs_helper.F90
  calc_rhs_helper-psc.f90
  calc_rhs.F90
  calc_rhs-psc.f90
  chease.F90
  chease-psc.f90
  checkpoint.F90
  checkpoint-psc.f90
  circular.F90
  circular-psc.f90
  collisions.F90
  collisions-psc.f90
  comm.F90
  comm-psc.f90
  debug_output.F90
  debug_output-psc.f90
  diag.F90
  diag-psc.f90
  dummyperf.F90
```

```
subroutine calc_rest(g_in,emfields_out,f_1_out)
  complex,dimension(li1:li2,lj1:lj2,lk1:lk2,ll1:ll2,lm1:lm2
  complex,dimension(lbx:ubx,lj1:lj2,lbz:ubz,lbv:ubv,lbw:ubw
  complex,dimension(lbx:ubx,lj1:lj2,lbz:ubz,1:n_fields),int

  call perfon('calcrest')

  if (x_local) then
     CALL field_solve_kxky(g_in,emfields_out)
  else
     CALL field_solve_xky(g_in,emfields_out)
  endif

  if (n_fields.eq.1) then
     if(perf_vec(2).eq.1) then
        call calc_f_es(g_in,f_1_out)
     else
        call calc_f_es_perf(g_in,f_1_out)
     endif
  else
     if(perf_vec(2).eq.1) then
        call calc_f_em(g_in,emfields_out(:,:,:,2),f_1_out)
     else
        call calc_f_em_perf(g_in,gy_pre,emfields_out(:,:,:,
     endif
  endif

  call exchange_5df(f_1_out)

  if (n_procs_v.gt.1) call exchange_v(f_1_out)
  if (collision_op.ne.'none') call exchange_mu(f_1_out)
```

```
SIR File: /home/rainy/workspaces/LRR/psctests/GEN
  subroutine: CALC_REST (54/220) (aux_fields-psc.
    call: FIELD_SOLVE_KXKY (166/166) (aux_fields
  subroutine: CALFULLRHS_KXKY_1 (40/119) (calc_
    call: MY_REAL_MAX_TO_ALL (79/79) (calc_rhs-
  subroutine: MY_REAL_MAX_TO_ALL (58/172) (con
    call: MPI_ALLREDUCE (114/114) (comm-psc.f9
  subroutine: MY_COMPLEX_SUM_VWSPEC (492/89
    call: MPI_ALLREDUCE (406/406) (comm-psc.f9
  subroutine: FIELD_SOLVE_KXKY (131/320) (field_
    call: MY_COMPLEX_SUM_VWSPEC (189/189) (f
  program: GENE (0/334) (gene-psc.f90:21)
    loop:  (0/334) (gene-psc.f90:147)
      userRegion:  (203/334) (gene-psc.f90:149)
        call: CALC_EXPLICIT_TIMESTEP (131/131)
  subroutine: CALC_EXPLICIT_TIMESTEP (142/145)
    loop:  (0/3) (time_scheme-psc.f90:122)
      call: CALC_REST (3/3) (time_scheme-psc.f9
```

Name	Filename	RFL	Severity	Region	Process	Thread
Stalls due to waiting for integer loads			70.09	Group		
L3 misses			70.01	Group		
IA64 Pipeline Stall Cycles			65.04	Group		
Stalls due to waiting for data delivery to register			62.56	Group		
Stalls due to branch misprediction flush			55.45	Group		
Stalls due to pipeline flush			47.89	Group		
L2 misses			34.36	Group		
L2 misses	aux_fields-psc.f90	42	49.20	CALL_REGION	46	0
L2 misses	ield_solve_kxky-psc.f9	37	46.61	SUB_REGION	83	0

Filter: Search: RE 0 Loaded - 88 Shown - 1 Selected -
Sort: [Severity (REV)]

Fig. 1.4: Snapshot of the Periscope GUI

Periscope and their summarized form, a multi-functional table is used for their visualization. To organize the displayed data, so that, maximum knowledge can be gathered out of it, the table provides the following features:

- Multiple criteria sorting algorithm
- Complex categorization utility
- Searching engine using regular expressions
- Filtering operations
- Direct navigation from the bottlenecks to their precise source location using the default IDE editor for that source file type (e.g. CDT/Photran editor).

An outline view for the instrumented code regions that were used in an experiment is also available. Statistical clustering is another key feature of the plug-in that enhances the scalability of the GUI and provides means of conducting peta-scale performance analysis. It can effectively summarize the displayed information and identify a runtime behavior possibly hidden in the large number of properties.

After every performance experiment of Periscope, the tool generates an XML-based file with the detected properties. This file, together with source code of the application and its SIR document, must be organized in a project for the Eclipse IDE so that the GUI can access it and load it for analysis. Because of the integration of the Eclipse File System (EFS) as a file management layer, the most appropriate data storage location can be freely chosen. As a result, a remote project can be created on a system where only SSH or FTP are available. Thus, there is no need to keep local and remote files synchronized and so it greatly enhances the whole tuning process of real-world applications.

1.4 Periscope Usage Model

This section explains the usage scenario of Periscope to find the performance bottlenecks in scientific applications.

1.4.1 Preparing Analysis Run

In order to undergo an analysis run, the user has to configure Periscope by stating the machine and the port number for the internal registry, analysis agents and applications. Optionally, the user can guide the search by stating the phase region via a user region in the code. For example, in Fortran-based applications, the user-region is marked as shown below:

```
!$MON USER REGION
        ...
!$MON END USER REGION
```

It is mandatory that the program has to be instrumented, compiled, and linked with the required libraries (MRIlib and mpiProfilerlib). In the existing application's makefile, the compilation step generating the object files has to be modified such that the compiler is replaced with the command *psc_instrument*. The script will preprocess the files, instrument them, and finally call the compiler for generating the instrumented object files. In addition, the compiler has to be replaced in the link step by *psc_instrument*. Here, it will link also the monitoring library to the executable as well as generate the SIR file with the program's static information.

Instrumentation of subroutines, call sites, user regions, loop regions, OpenMP parallel regions can be switched on/off using the keywords *sub, call, user, loop,* and *par*. This specification can be given for each source file individually via the *psc_config_file* of Periscope.

1.4.2 Starting an Analysis Run

For starting an analysis run, the user has to start a registry service by specifying its alloted port number. On the Altix, the following command is used to start the registry:

```
regsrv.ia64 35000 &
```

The agents and the application processes register with it their location and the ports they use. The Periscope Frontend starts the analysis agent hierarchy. It also runs the application and optimizes the mapping of application processes and agents to processors. The number of processes and threads are specified via *ompnumthreads* and *mpinumprocs* arguments. It will first contact the registry and then start the application. After all application processes registered with the registry, the agent hierarchy starts, the analysis agents connect to the application processes and the search starts. The following command is used for starting an analysis run on our Altix:

```
frontend.ia64 --apprun=~/psctest/GENE/gene
      --mpinumprocs=1024 --strategy=
       StallCycleAnalysis --debug=1
```

The *apprun* parameter specifies the command line to start the application. It will be passed to the mpirun command. Specifying the number of MPI processes is done using the command *mpinumprocs*. The argument *strategy* can be one of the following: MPI, StallCycleAnalysis, StallCycleAnalysisBreadthFirst, and AllStrategy. The command *debug* is for setting the debug level. In addition, the OpenMP threads, SIR filename, application port number, number of processors controlled by a single analysis agent, number of highlevel agents and the timeout can be explicitly specified in the command line.

1.5 Distributed Performance Analysis

As an example for identifying performance bottlenecks with Periscope for a large-scale scientific application, we demonstrate here the performance analysis of a Gyrokinetic Electromagnetic Numerical Experiment (GENE) code. The GENE code [2, 8] is an iterative solver for a non-linear gyrokinetic equations in a 5-dimensional phase space to identify the turbulence in magnetized fusion plasmas. It was developed in the Max Planck Institute for Plasma Physics in Garching. GENE has the capability to run with a large number of processors, such as, 512 or 1024. It consists of 47 source files with 16,258 lines. The code is written in Fortran 95 with MPI-based parallelization.

The experiments were carried out on the ALTIX at LRZ. In the test, the application was executed in eight partitions with 134, 154, 24, 144, 50, 257, 143, 94 processors and one analysis agent in each partition. The GENE code was analyzed on 1, 8, 16, 32, 64, 128, 256, 512 and 1024 processors based on its parallelization needs.

The experimental results revealing the found properties in different code regions of GENE, scalability issues that include single node performance, loading properties, organizing information, and clustering mechanisms are detailed in the following subsections.

1.5.1 Found Properties

The main program, gene.f90 of 47 files is responsible to read the parameters, initialize conditions and calculate the explicit time loops. The user-region indicating the phase for the incremental analysis was marked covering the time step.

The properties found are summarized in Table 1.1. The column headings with 1, 2, 3, 4, and 5 represents the properties, respectively, IA64 pipeline stalls, Stalls due to L1D TLB misses, Stalls due to pipeline flush, Stalls due to waiting for FP registers, and Stalls due to waiting for integer registers. It can be seen that the code suffers more from integer loads compared to floating point registers.

Table 1.1: Found properties in GENE code

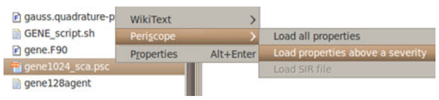
Sl.No	File Name	Region	Line Number	Average Severity				
				1	2	3	4	5
1	gene.f90	User Region	149	38.3	6.85	11.1	4.14	17.65
		Call Region	163	37.6	4.41	10.2	3.0	20.4
2	time_scheme.f90	Subroutine	91	37.7	3.1	10.3	2.4	22.2
		Loop Region	122	22.2	*	6.0	2.8	11.1
		Call Region	129	12.4	*	7.7	*	13.2
3	field_s_kxky.f90	Subroutine	37	18.5	*	11.9	*	19.3
		Call Region	65	17.7	*	9.6	*	19.6
3	aux_fields.f90	Subroutine	34	18.3	*	11.5	*	19.8
		Call Region	42	17.9	*	11.9	*	19.4
3	comm.f90	Subroutine	305	17.5	*	11.9	*	19.6
		Call Region	312	17.1	*	1.1	*	*

1.5.2 Scalability of the Analysis

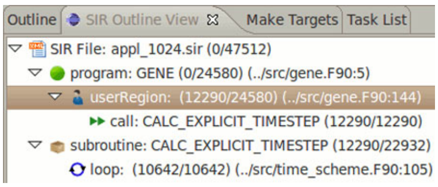
Several techniques were introduced in Periscope to support the analysis of large-scale applications.

1.5.2.1 Loading Properties

While analyzing the GENE code with 1024 CPUs, Periscope detected plenty performance properties of 14 different types which were distributed in 4 code regions. In order to enhance the analysis an additional menu entry (see Fig. 1.5a) is provided in the GUI to load only properties with high severity.

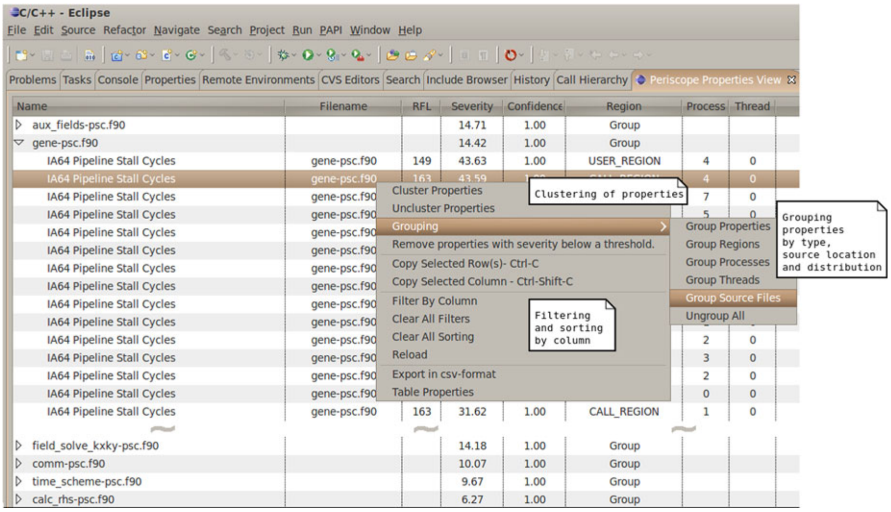


(a) Loading properties above a threshold



SIR File: appl_1024.sir (0/47512)
program: GENE (0/24580) (../src/gene.F90:5)
userRegion: (12290/24580) (../src/gene.F90:144)
call: CALC_EXPLICIT_TIMESTEP (12290/12290)
subroutine: CALC_EXPLICIT_TIMESTEP (12290/22932)
loop: (10642/10642) (../src/time_scheme.F90:105)

(b) Instrumentation nesting


C/C++ - Eclipse
File Edit Source Refactor Navigate Search Project Run PAPI Window Help
Problems Tasks Console Properties Remote Environments CVS Editors Search Include Browser History Call Hierarchy Periscope Properties View
Cluster Properties
Uncluster Properties
Grouping
Remove properties with severity below a threshold.
Copy Selected Row(s)- Ctrl-C
Copy Selected Column - Ctrl-Shift-C
Filter By Column
Clear All Filters
Clear All Sorting
Reload
Export in csv-format
Table Properties
Group Properties
Group Regions
Group Processes
Group Threads
Group Source Files
Ungroup All
Clustering of properties
Grouping properties by type, source location and distribution
Filtering and sorting by column

Name	Filename	RFL	Severity	Confidence	Region	Process	Thread
aux_fields-psc.f90			14.71	1.00	Group		
gene-psc.f90			14.42	1.00	Group		
IA64 Pipeline Stall Cycles	gene-psc.f90	149	43.63	1.00	USER_REGION	4	0
IA64 Pipeline Stall Cycles	gene-psc.f90	163	43.59	[illegible]	[illegible]	4	0
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	7	0
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	5	0
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	2	0
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	3	0
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	2	0
IA64 Pipeline Stall Cycles	gene-psc.f90	[illegible]	[illegible]	[illegible]	[illegible]	0	0
IA64 Pipeline Stall Cycles	gene-psc.f90	163	31.62	1.00	CALL_REGION	1	0
field_solve_kxky-psc.f90			14.18	1.00	Group		
comm-psc.f90			10.07	1.00	Group		
time_scheme-psc.f90			9.67	1.00	Group		
calc_rhs-psc.f90			6.27	1.00	Group		

(c) Grouping and clustering the information

Fig. 1.5: Customizing the displayed information

1.5.2.2 Organizing Information

Processing the large amount of properties from a large-scale execution might be a challenging job for the application developers. In order to tackle this problem, Periscope provides a way to group the collected data in different categories according to the type and the location of the performance property as shown in Fig. 1.5c. Furthermore, the displayed data can be filtered from the instrumentation outline view to show only the entries of a selected code region. The view also exposes the complete instrumentation nesting (see Fig. 1.5b).

1.5.2.3 Clustering Mechanism

The GUI for Periscope provides a basic multivariate statistical clustering support that is based on a data-mining workbench called Weka. This feature is activated from the context menu as shown in Fig. 1.5c. Currently, we used the *SimpleKMeans* algorithm to group the detected performance bottlenecks based on their distribution on the CPUs and their respective code regions. During the analysis of the GENE code with 1024 processors, the algorithm generated a few clusters of similarly behaving processors. It scaled down the amount of displayed information and so made it easier to uncover runtime behavior that was previously hidden in the huge dataset.

1.5.2.4 Single Node Performance

The single node performance for large-scale run (Fig. 1.6) illustrates the efficiency of the computation. For the run of GENE with 1024 processors, Periscope identified that processor 112 had an unexpected peak of 13 severity points from its average for IA64 pipeline stalls, and processor 868 had 19 severity points for stalls due to L1D TLB misses. The figure shows that the processors 834 to 862 had comparatively good performance due to less severity than other processors.

1.6 Conclusion and Outlook

Periscope is an online-based performance analysis tool for detecting performance problems in large-scale scientific applications. The search is executed in an incremental fashion by either exploiting the repetitive behavior of program phases or by restarting the application several times. The GUI provides an excellent user-friendly interface for the developers.

This article presented the agent design to find the wide range of bottlenecks, Periscope's GUI with excellent features, such as, remote execution, grouping large datasets for scalability issues, and clustering of found properties. In addition, the tool was demonstrated with the real-world large-scale scientific application, namely,

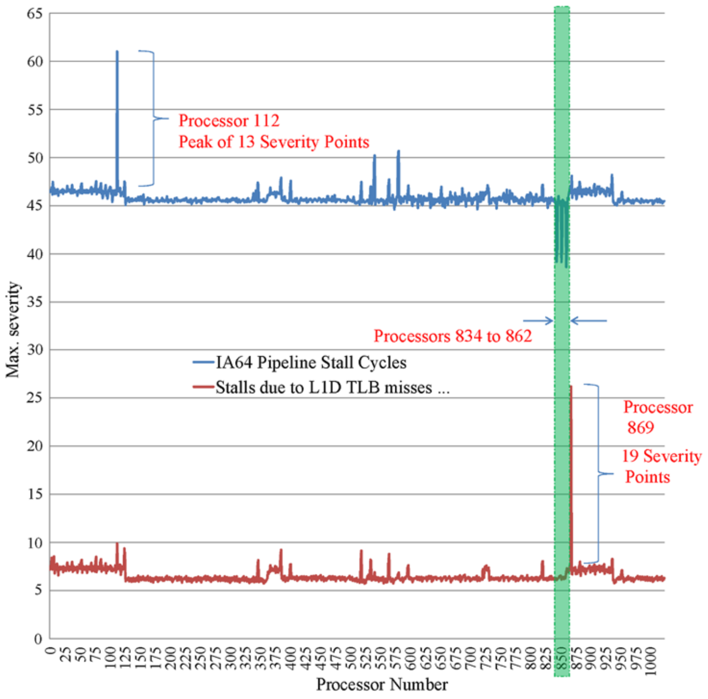


Fig. 1.6: Single node performance of GENE when running in Altix for 1024 processors

GENE to find the single node performance on large-scale runs, its scalability issues and the found properties. Since the analysis was carried out with the Periscope GUI, the tool was found to be user-friendly and efficient for displaying the necessary information.

While the largest test runs with Gene were executed for 1024 processes due to the availability of test data for that size, we did other experiments with up to 4000 processors on the Altix. These experiments helped us in improving the scalability of Periscope considerably.

Periscope is currently running on Itanium Altix, Power 6, and Linux X86-based parallel machines. In the near future, we plan to port Periscope to BlueGene/P.

References

1. Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In *Proc. of the 2nd Int. Work. on Parallel Tools for HPC, HLRS, Stuttgart*, pages 139-155, Springer Publications, July 2008.

2. Chen, Y. and Parker, S. E. A δf particle method for gyrokinetic simulations with kinetic electrons and electromagnetic perturbations. In *Comput. Phys. 189, 2 (Aug. 2003)*, DOI: http://dx.doi.org/10.1016/S0021-9991(03)00228-6. pages 463-475, 2003.
3. Eric Clayberg and Dan Rubel. Eclipse Plug-ins. In *Addison-Wesley Professional, ISBN 978-0-321-55346-1* pages 107-135, 2008.
4. Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In *Proc. of the 13th Eur. PVM/MPI Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2006), pages 303–312, Bonn, Germany, 2006*.
5. M. Gerndt and K. Fürlinger. Specification and detection of performance problems with ASL. *Conc. and Computation: Prac. & Exp.*, 19(11):1451–1464, Aug 2007.
6. Michael Gerndt and Edmond Kereku. Search strategies for automatic performance analysis tools. In Anne-Marie Kermarrec, Luc Boug, and Thierry Priol, editors, *Euro-Par 2007*, volume 4641 of *LNCS*, pages 129–138. Springer, 2007.
7. Jeffrey Vetter and Chris Chambreau. mpiP: Lightweight, Scalable MPI Profiling. http://mpip.sourceforge.net, 2008.
8. F. Jenko. Massively parallel vlasov simulation of electromagnetic drift-wave turbulence. In *Comp. Phys. Comm. 125* 2000.
9. B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer, Vol. 28, No. 11, pp. 37-46*, 1995.
10. Philip C. Roth and Barton P. Miller. The distributed performance consultant and the sub-graph folding algorithm: On-line automated performance diagnosis on thousands of processes. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, March 2006.
11. Shajulin Benedict, Matthias Brehm, Michael Gerndt, Carla Guillen, Wolfram Hesse and Ventsislav Petkov. Automatic Performance Analysis of Large Scale Simulations. In *PROPER 2009, (in press), Springer Publishers* 2009.
12. Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.
13. Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 13–22. IEEE Computer Society, February 2003.