Hong-Linh Truong

Distributed and Parallel Systems Group, Institute for Computer Science,

University of Innsbruck, *truong@dps.uibk.ac.at*

# Novel Techniques and Methods for Performance Measurement, Analysis and Monitoring of Cluster and Grid Applications

– PhD Dissertation – Vienna University of Technology

Last revised: 18th April 2005

TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

DISSERTATION

# Novel Techniques and Methods for Performance Measurement, Analysis and Monitoring of Cluster and Grid Applications

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften unter der Leitung von

o. Univ.-Prof. Dipl. Ing. Dr. Thomas Fahringer
Institut für Informatik, Leopold-Franzens Universität Innsbruck

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

**Dipl. Ing. Hong Linh Truong**
Matrikelnummer: 0027858
Forsthausgasse 2-8/2302, A-1200, Wien

Wien, am 28. Januar 2005

TO MY PARENTS
AND HADI
FOR THEIR PATIENCE AND UNDERSTANDING

# ABSTRACT

The evolution of parallel and distributed architectures and programming paradigms for performance-oriented program development challenges the state of technology for performance tools. Coupling different programming paradigms such as message passing and shared memory programming for hybrid cluster computing (e.g. SMP clusters) is one example for high demands on performance analysis tools that are capable to cope with applications for multiple programming models and target architectures. Performance tools must be able to observe performance problems at all levels of a system while relating low-level behavior to the application program.

This dissertation presents a set of novel techniques and methods for performance instrumentation, measurement, monitoring and analysis of cluster and Grid applications. We introduce a classification of temporal overheads for parallel programs that can be used to explain sources of performance problems. A highly customizable instrumentation and measurement system allows to control the instrumentation and performance measurement for code regions and performance metrics of interest in a flexible and automatic way. A data repository is employed in order to store performance data and information about performance experiments which alleviates the association of performance information with experiments and the source code. Performance analysis can be conducted for single and multiple experiment(s).

To overcome the limitation of existing performance analysis techniques, which are based on the hard computing model, we propose a novel approach to performance analysis for parallel and distributed systems that is based on soft computing. Firstly, we introduce the concept of performance score representing the performance of code regions that is based on fuzzy logic. We then propose techniques for classifying the performance according to fuzzy terms representing performance characteristics. A high-level query language is introduced to support the search for performance problems by using linguistic expressions. We present fuzzy-based ranking analysis and bottleneck search. Secondly, we propose methods for measuring performance similarity among code regions and among experiments. We introduce the performance similarity analysis that can be employed for multiple experiments. Finally, we develop and implement a fuzzy C-means clustering for performance analysis, and introduce fuzzy rules that can be utilized to filter irrelevant performance data. We propose the soft performance analysis approach in order to support making soft decisions on evaluation, classification, search and analysis of the performance of parallel and distributed programs, rather than only hard decisions as in most existing performance tools.

This dissertation introduces a unified monitoring and performance analysis system for the Grid. The unified system integrates performance monitoring, dynamic instrumentation of Grid applications, and performance analysis of

Grid scientific workflows into a single system, which is implemented as a set of grid services based on the Open Grid Services Architecture (OGSA). The unified system provides an infrastructure for conducting online monitoring and performance analysis of a variety of Grid services including computational and network resources, and Grid applications. We develop a self-managing sensor-based middleware for monitoring and integrating performance data in Grids. The middleware unifies both system and application monitoring in a single system, storing and providing a variety of types of monitoring and performance data in decentralized locations. We have developed event-driven and demand-driven sensors to support rule-based monitoring and data integration. Grid service-based operations and TCP-based data delivery are exploited in order to balance tradeoffs between interoperability, flexibility and performance. Peer-to-peer features have been incorporated into the middleware, enabling self-configuring capabilities, supporting group-based and automatic data discovery, data query and subscription of performance and monitoring data.

We present Grid services for dynamic instrumentation of Grid-based applications, performance monitoring and analysis of Grid scientific workflows. We describe a Grid service to support dynamic instrumentation of Grid applications. The dynamic instrumentation service provides a widely accessible interface for other services and users to conduct the dynamic instrumentation of Grid applications during the runtime. We introduce a Grid performance analysis service for Grid scientific workflows. The analysis service utilizes various types of data including workflow graphs, monitoring data of resources, execution status of activities, and performance measurements obtained from the dynamic instrumentation of invoked applications. We store workflows and their relevant information including performance metrics, devise techniques to compare the performance of constructs of different workflows, and support multi-workflow analysis.

Finally, we propose a new approach to performance analysis, data sharing and tool integration in Grids that is based on ontology. We devise an ontology for describing the semantics of monitoring and performance data that can be used by performance monitoring and measurement tools. We introduce an architecture for ontology-based performance analysis, data sharing and tool integration. The core of this architecture is a Grid service which offers facilities for other services to archive and access ontology models along with collected performance data, and to support the search and reasoning about performance data.

As a proof-of-concept, we have developed SCALEA and SCALEA-G which implement the above-mentioned techniques and methods. This dissertation presents several experiments of real-world applications and examples to validate the proposed methods and techniques.

# ACKNOWLEDGMENTS

*This dissertation cannot be finished without the involvement of a number of people*:

First of all, I would like to especially thank Professor Thomas Fahringer, who over the years has been an inspirational supervisor. He first brought me to Austria and has tirelessly provided supports and guidance for my PhD study. I am grateful to him for guiding my research career and leading me to new research challenges. I also would like to acknowledge the critical comments and suggestions he has given me to improve my weakness. My knowledge has been enriched by his judgment and wisdom.

Professor Schahram Dustdar, my second advisor, has provided me the guidance for the study on the impact of peer-to-peer computing on workflow management systems that eventually has impacted my research greatly. I would like to thank him for valuable comments and suggestions that help me to improve the quality of this dissertation.

Over the past four years, the Austrian Science Fund (FWF) and the Institute for Software Science (IFS) of the University of Vienna have supported my work within the framework of the Special Research Program AURORA. The EU projects, IST APART and IST K-WF Grid, have partially supported my work. My research cannot be carried out without these fundings.

I would like to express my thanks to Professor Hans Zima, the chair of IFS and the speaker of AURORA, and to Professor Siegfried Benkner, the head of IFS for their continuous supports. I really appreciate the help of Dr. Ian Glendinning, who has helped me to improve scientific writing styles. I am grateful for the helps and co-operation of many AURORA researchers, who have provided their applications for the experiments conducted in this dissertation, and of the system administrators of IFS and AURORA, who have helped to establish the environment for these experiments. Many thanks are to the secretaries of AURORA and IFS, namely Elisabeth Wurth, Elisabeth Obermaier and Marry Cherry, for their kind helps and assistance.

Over the course of my PhD research, my colleagues in the tool group, Alexandru Jugravu, Sabri Pllana, Radu Prodan and Clovis Seragiotto, have not only provided valuable discussions in our research fields but also brought up many interesting social issues. The pleasant time working in a group of international researchers with different cultures has led to many memories that would never be forgotten.

I am in debt to my loving wife, Thu Ha. Her love and deeply understanding have helped me to overcome the most difficult moments, encouraging me to forget the bad things and giving me a new hope when I am down. She not only nourishes me with the best Vietnamese foods and but also takes care for every tiny things of my life. Last but not least, I deeply thank my parents for always supporting and believing in whatever I do. Without their love and supports, I would never be able to finish this dissertation.

Vienna, 28th January 2005

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Performance analysis of distributed and parallel programs is a complicated process which involves a variety of computer science and engineering fields. Performance analysis is a required task for developing efficient and scalable parallel and distributed applications. Performance analysis not only helps the application developer to assess and improve the performance of applications, to identify potential performance problems, and to determine locations in the code where the performance problems occur but also provides essential feedbacks to the design and implementation of system software and middleware (e.g., operating system, communication library and resource broker), programming models (e.g., message passing, thread-based and data parallel), computer architectures and communication networks. Therefore, performance analysis tools have a very important role in developing effective, efficient and high performance applications and computer systems.

Over the past decades, a large number of architectures for high performance computers have been introduced such as vector computer [43, 76, 210, 152, 153], MPP [198, 259], SMP [181, 185, 69], etc. With the continuous advances of novel high performance microprocessors and the existence of high speed networks as off-the-shelf commodity components, clusters of SMPs or PCs/workstations appear to be a promising solution to design low cost parallel machines [186, 53].

The trend of using cluster of SMP requires programming efforts for porting message passing applications to the hybrid memory model. On the one hand, a large number of existing high performance applications still use the message passing model. On the other hand, parallel applications have shifted from thread-aware, tuple space, message passing, HPF [127] to OpenMP [266], hybrid parallel models such as OpenMP/MPI. MPI [124] has been developed for parallel programming of distributed memory multiprocessor systems. OpenMP has recently emerged as the standard for parallel programming of SMP architectures. Using MPI for handling communication between SMP nodes and OpenMP for implementing parallelization strategy within an SMP node, the hybrid OpenMP/MPI paradigm is the emerging parallel programming model for the development of parallel applications on current SMP clusters [125, 224, 57, 123, 229].

Recently, Grid computing [99, 41] has presented a compelling vision of using geographically and institutionally computational and information resources that is based on the concept of virtual organization [147]. Grid computing aims to integrate and coordinate diverse and disparate computation, data, and other resources and to provide a uniform access to them. Despite the tremendous potential to the Grid paradigm, as well as large investments on and strong commitments to the research of Grid computing, the dynamic and complex nature of the Grid environment poses many challenges that have not been solved yet.

The evolution of parallel and distributed architectures and programming paradigms for performance-oriented program development challenges the state of technology for performance tools. Coupling different programming paradigms such as message passing and shared memory programming for hybrid cluster computing (e.g SMP clusters) is one example of high demands on performance analysis tools. Performance analysis tools should be able to cope with applications for multiple programming models and target architectures. Performance tools must be able to observe performance problems at all levels of a system while relating low-level behavior to the application program. As Grids evolve from clusters to virtualized organizations deployed in distributed and wide-area systems, the monitoring and performance analysis for the Grid requires standards and many novel techniques to deal with challenges posed by Grid environments. Few software tools exist to assist the user to monitor and analyze the performance of Grid-enabled applications, especially scientific workflows in the Grid.

To keep the pace with the evolution of state-of-art parallel and distributed architectures and models (e.g., from SMP to clusters of SMP to Grid systems) and of parallel and distributed applications (e.g., from MPI, OpenMP to OpenMP/MPI to Grid workflows), effective instrumentation and measurement techniques for these applications, systems and models need to be investigated and developed. Due to the complexity of the applications and the systems on which the applications are executed, there is a need of collecting, gathering and utilizing monitoring and performance data from many sources, which may be distributed and diverse, in order to understand the performance of the applications. As a result, techniques for integrating monitoring and performance data are important. The complexity and quantity of performance measurements are so overwhelming that new performance analysis techniques are required to support efficient, scalable and fast analyses. In addition, performance data needs to be shared and exchanged among different tools. Therefore, techniques and methods to represent and archive performance data and to support tool integration are of prominent importance.

## 1.1 Motivation

Performance analysis for parallel and distributed computing is faced with numerous complexities and open problems that have not been solved by existing tools or technologies. These open problems or shortcomings of existing approaches motivated the research described in this dissertation.

*How to determine program locations that expose performance problems and to explain to the user the performance problems?*: Commonly, the gap between ideal and measured execution time reflects the performance problems in user's programs. This gap is commonly defined as the overhead implied by parallelizing a program. A classification of performance overheads will provide the user with a detailed understanding of where and how performance is lost. As programming models constantly being evolved, the classification has to be updated to cover (new) performance overheads of the new programming models. Therefore, a classification of performance overheads for message passing, shared memory and hybrid parallel programs should be investigated. Most existing tools, however, lack a classification of performance overheads and only support limited measurement and instrumentation features, thus a systematic performance analysis is severely hampered.

*The realization that one of the challenges faced by performance analysis tools is the selection of an appropriate level of measurement and analysis detail in order to systematically identify the origins of performance problems*: A programmer may be well aware of which code sections and performance metrics should be in the center of the performance analysis. On the other hand, tools can automatically divide a program into code regions, determine whether those regions cause performance problems or not, and map code region to a given class of performance overheads. A combination of user directions and tools functionality can provide a simple and efficient mechanism to control the level of measurement and analysis. Most existing instrumentation systems do not provide enough flexibility for the user to specify code regions for which performance metrics of interest should be determined. Mostly, they just allow the user to specify a set of pre-defined code regions with built-in performance metrics.

*The need of supporting the user in conducting performance analysis across experiments*: In the cycle of application development, the developer refines and develops several code versions, and then conducts several experiments by running programs on various platforms. The resulting experiments are used not only for verifying the correctness of the program but also for the performance comparison. Unfortunately, there is a lack of tools that are able to compare and analyze multiple experiments. Most users have to manipulate performance data of experiments in order to perform the comparisons. Besides, many experiments generate a huge amount of performance data which further needs to be filtered and archived in order to support the performance analysis and comparison of multiple experiments. This large volume of performance data can also be processed by other performance analysis tools, which can provide more analyses for the user, as well as can be used by high-level tools. To date, little effort has been done to employ data repositories to organize and store performance data for performance analysis. In addition, different tools represent performance data in different ways, and they lack the capability to export their data to other common formats. Consequently, the collaboration between performance tools and high-level tools is still a difficult task.

*The lack of specification and control of inexact parameters, commands and requests in existing performance analysis tools*: Current techniques in existing performance analysis tools have mainly been used to process the performance data that are in the form of precise numerical and measurement-based data. Firstly, these techniques always apply *exact* analysis methods that result in exact conclusions about performance characteristics of applications. However, performance measurements collected may have limited accuracy or missing data, in other words, performance measurements may be incomplete and uncertain data. Therefore, exact analysis methods might not be appropriate for processing uncertainty data. Secondly, performance tools based on numerical analysis interact with the user through complex numerical values which are not easily understood by the end user. Moreover, in the real world we largely rely on domain expertise and user-provided inputs as parameters to control the performance analysis and tuning. Such expertise and inputs may be inexact and uncertain. However, there is no mechanism to specify and control approximate and inexact parameters in existing performance analysis tools, in other words, these tools do not provide a mechanism for making soft decisions. These techniques are based on *hard computing* models, which are based on binary logic, crisp system and numerical analysis and do not accept imprecision and uncertainty. To address the issues mentioned above, we can investigate performance analysis techniques that are based on soft computing models, e.g. fuzzy logic and machine learning, which are useful tools for processing and analyzing uncertainty and large-volume data. *Soft computing* [278, 45, 202] presents another way to evaluate and analyze data that is based on the concept of *soft, inexact, uncertainty* about the data. Unlike hard computing, soft computing accepts imprecision and uncertainty, supports approximate reasoning, offers cost effectiveness, and is much closer to the real world [278, 45]. By integrating soft computing techniques, we can develop techniques for performance tools that can (i) extract useful performance information from large, dynamic and multi-relational performance measurement sources, (ii) support the specification and the control of approximate and inexact parameters, commands and requests in existing performance analysis tools, and (iii) interact with the user through high level notions and concepts expressed in linguistic expressions.

*The need of a unified performance monitoring and analysis system for the Grid*: Grid monitoring is a crucial task that provides useful information for several functions such as performance analysis and tuning, performance prediction, fault detection and scheduling. However, most existing Grid monitoring tools are separated into Grid system monitoring and Grid application monitoring. The lack of the combination of two domains in a single tool makes the user difficult to correlate performance metrics provided by various tools at different levels of representation when performing Grid monitoring and performance analysis. Grid performance tools that combine application- and system-specific monitoring and performance analysis will provide users a unified view and support users to correlate performance metrics from various sources. Moreover, although recently the Grid evolves towards a Grid system architecture based on service-oriented concepts, existing performance analysis tools are not service-oriented yet. Due to the fact that they lack a set of interfaces and conventions that follow specifications of Grid/Web services, they are not well suited for Grid environments. A Grid performance tool is used to conduct monitoring and performance analysis for the Grid, thus, it must address and deal with the challenges in the Grid such as geographically dispersion, heterogeneity, security, etc. Grid computational, software, and network services should be monitored and related together in a single system. Therefore, the Grid performance tool has to integrate instrumentation, measurement, and analysis and monitoring of a variety of Grid services at various levels of abstraction.

*The lack of Grid services for dynamic instrumentation, monitoring and performance analysis of scientific workflows in Grids*: even though many existing tools support the construction and execution of scientific workflows in the Grid. While existing Grid toolkits provide core services for job submission, resource discovery, similar Grid services for instrumenting Grid applications do not exist. The instrumentation of Grid application must be carried out by the end user. Consider the diversity and dynamics of the Grid sites. On the one hand, if a user wants to instrument his code, the user has to know in advance the Grids he submits job to, and has to select the right instrumentation tool for a Grid site. As a result, the user has to do a daunting task in order to instrument his code. Moreover, the selected instrumentation tool may not work with the monitoring middleware deployed in the selected Grid site. On the other hand, instrumentation techniques are typically bound to specific languages and systems. More importantly, workflows tend to be composed from deployed components whose source code is not available. Without the instrumentation of code regions of workflow activities themselves, we are only able to monitor at the level of activity, thus it significantly reduces the ability to detect and correlate performance problems. Recently, increased interest can be witnessed in exploiting the potential of the Grid for scientific workflows. As the Grid is diverse, dynamic and inter-organizational, even for a particular scientific experiment, there is a need of having a set of different workflows because (i) one workflow mostly suits for a particular configuration of the underlying Grid environment, and (ii) the available resources allocated for a scientific experiment and their configuration are changed for every different run on the Grid. This requirement is a challenge in the performance monitoring and analysis of workflows because very often the client of the performance monitoring and analysis tools wants to compare the performance of different workflow constructs with respect to the resources allocated in order to determine which workflow construct should be best mapped to which Grid site. Therefore, multi-workflow analysis, the analysis and comparison of the performance of different workflow constructs, ranging from the whole workflow to a specific construct (e.g. a fork-join construct), is an important feature.

*How to understand and process performance data where the data is diversely collected and when there is no central component to manage and provide its semantics?*: This question is raised by the Grid community. Normally, users run their applications in multiple Grid sites, each is equipped with different computing capabilities, platforms, libraries that require different performance tools to conduct the monitoring and measurement. A single performance tool no longer meets user's need. In the absence of the central component, performance monitoring and measurement tools need an agreed and well-defined semantic representation for describing the data they collect and provide because on top of them many tools and services need performance data for further works such as performance analysis, scheduling and resource matching. Current Grid performance tools focus on the monitoring and measurement, but neglect the representation for performance data. A widely accepted semantic representation of data helps in sharing the data, leveraging the tool integration and reducing the human intervention in the performance analysis process. However, due to the complexity of the Grid and the diversity of resources in Grid, any proposed data representation should easily be extended to describe new resources and metrics. To that end, we can devise a novel ontology for representing semantics of performance data that can be used by performance monitoring and measurement tools. Based on that ontology, a Grid service can be developed to archive and provide ontology models with collected performance data to performance analysis tools and other services.

Stimulated by above motivating factors, in this dissertation we propose solutions to address the above-mentioned drawbacks.

## 1.2 Contributions

In the following we describe the contributions of this dissertation to the shortcomings of existing approaches as mentioned in the previous sections.

The first contribution is **a novel classification of temporal overheads and overhead analysis techniques** for shared and distributed memory parallel programs that includes data movement, synchronization, control of parallelism, additional computation, loss of parallelism, and unidentified overheads.

The second contribution is **a customizable instrumentation and measurement system** which can be precisely controlled by program directives and command-line options. The user can specify code regions for which performance metrics should be determined. Our instrumentation system is one of the first that can support a variety of program paradigms including sequential, MPI, OpenMP, HPF and hybrid parallel programs (OpenMP/MPI, OpenMP/HPF).

The third contribution is **the implementation of an experiment data repository for performance analysis, data sharing, tools integration** and **multi-experiment analysis**. We employ a performance data repository to store the most important information about performance experiments including application, source code, machine information, and performance results.

The fourth contribution is **the soft performance analysis approach**. We propose a new approach to performance analysis that is based on soft computing. In this approach, soft computing techniques, e.g. fuzzy logic, machine learning, and the combination of the two, are utilized for evaluating the performance of parallel and distributed applications. We introduce flexible and convenient methods to deal with uncertainty in performance analysis, e.g. fuzzy-based bottleneck search, and means for conducting performance analysis in the form closer to human notions, e.g. fuzzy-based search query.

The fifth contribution is **the design and implementation of a novel unified monitoring and performance analysis system for Grids**. We have designed and implemented a sensor-based middleware for performance monitoring and data integration. The middleware is capable of self-management, supporting event- and demand-driven sensors, rule-based monitoring. Based on that middleware a unified monitoring and performance analysis system that unifies Grid infrastructure and application monitoring, source code and dynamic instrumentation, and performance analysis into a single system.

The sixth contribution is **the design and implementation of a novel Grid dynamic instrumentation service for Grid-based applications**. We develop a Grid service that supports the dynamic instrumentation of Grid applications. We devise a novel instrumentation request language for facilitating the interaction between instrumentation service and clients during the instrumentation of applications in Grids. We develop a standardized intermediate representation for binary code which can be used to describe the application structure.

The seventh contribution is centered around **novel techniques and methods for performance analysis of scientific workflows in the Grid**. We have developed an integrated system to support performance monitoring and analysis of scientific workflows in Grids. We introduce techniques to support the performance analysis of multiple workflows.

The eighth contribution is **a novel ontology for describing performance data and a Grid service of ontology-based performance data repository**. Initial results show that ontology approach is a promising solution in the domain of performance analysis because it not only provides a common understanding about performance data among tools but also increases the automation of performance analysis.

As a result of our research, we have implemented several novel methods and techniques for instrumentation, measurement, monitoring and performance analysis of cluster and grid applications in two frameworks named SCALEA [246] and SCALEA-G [249]. SCALEA includes an instrumentation system, measurement, performance analysis and visualization tool for parallel programs. SCALEA-G is an integrated framework that supports monitoring and performance analysis of Grid systems and applications, dynamic instrumentation of Grid applications, and the integration of multiple types of monitoring and performance data.

The work described in this dissertation has been widely accepted in the area of performance analysis, measurement and monitoring for cluster and Grid architectures. This dissertation is based on the papers which published contributions described above [255, 244, 254, 245, 246, 251, 248, 250, 249, 252] together with the papers submitted for publication [253, 247].

## 1.3 Dissertation Overview

The rest of this dissertation is organized as follows: in Chapter 2 we briefly present the related work and outline research challenges.

Chapter 3 presents preliminaries for the rest of this dissertation. We discuss parallel architectures, the Grid, and their supported programming models, and other related concepts such as program structure and code regions, and performance experiment model. We present a new data structure named dynamic code region call graph which reflects the dynamic relationship between code regions and their subregions and enables a detailed overhead analysis for code region.

Chapter 4 presents an overview of the SCALEA and SCALEA-G framework. We introduce SCALEA, a performance instrumentation, measurement, and performance analysis for parallel programs, and SCALEA-G, a unified performance monitoring, instrumentation and analysis system for the Grid.

Chapter 5 details novel techniques and methods for performance measurement, instrumentation and analysis for cluster applications. We introduce a classification of temporal overhead, the overhead analysis and a customizable instrumentation system. We present an experiment data repository for performance analysis and its utilization. We introduce a novel approach to performance analysis for parallel and distributed systems that is based on soft computing.

Chapter 6 details novel techniques and methods for performance instrumentation, monitoring and analysis of the Grid. We present a sensor-based middleware that unifies the performance monitoring and analysis for Grid applications and infrastructures in a single framework. We introduce a Grid service for dynamic instrumentation of Grid applications. We present the techniques for monitoring and performance analysis of Grid scientific workflows.

A novel approach to performance analysis that is based on ontology is introduced.

Chapter 7 presents experiments of real-world applications as a proof of concept of our methods and techniques. We present several experiments conducted on cluster and Grid environments to demonstrate the usefulness of our proposed methods and techniques.

Chapter 8 summarizes the dissertation and gives an outlook to the future work.

# 2

# Related Work

## 2.1 Introduction

Our work is best described as an attempt to develop a set of novel techniques and methods for instrumentation, measurement, monitoring and analysis of parallel and distributed programs on cluster and Grid systems. In this chapter, we discuss the related work in the following topics:

- *instrumentation and measurement techniques*
- *performance data representation and repository*
- *overhead analysis for parallel programs*, *multi-experiment analysis*, *soft performance analysis*
- *monitoring and performance analysis for the Grid*

## 2.2 Performance Measurement and Instrumentation

Instrumentation techniques have a long history. Several techniques can be used to instrument distributed and parallel programs. Instrumentation can be employed at many levels such as source code, object code, libraries and binary code, and at static, before the execution of the application, or dynamically, during the runtime of the application (for a greater detail, see [105, 222]). Our work is mostly related to the automatic instrumentation at source code level for parallel programs and to the dynamic instrumentation of Grid applications. In the following, we briefly describe instrumentation approaches and related tools.

### 2.2.1 Customizable Instrumentation

For *source code instrumentation*, instrumentation probes, which are used to measure the performance of code regions, are inserted into source files automatically by an instrumentation system or manually by the end-user. In an automatic source code instrumentation system, the system, maybe under the guidance of the user or the high-level tool, will insert instrumentation probes directly into source files of the program. The key elements of an automatic instrumentation system are a mechanism for constructing a parsing tree from source code, a set of functions for manipulating and restructuring the parsing tree, and an unparsing mechanism for generating new source code from the restructured form. Some automatic instrumentation systems are based on parser and unparser part of a compiler infrastructure; others just use simple text parser and unparser that are independently from any compilers. The advantages of source code instrumentation are i) we can instrument at arbitrary level of source codes (e.g. loop, function call), and ii) we can relate performance data to the source code. However, this approach requires the availability of source code. Especially, in the case the automatic instrumentation is carried out with the support of compiler parser and unparser, this approach may require all source modules; pre-compiled module may not be accepted.

Several tools follow the source code instrumentation approach. TAU [221] performance framework is an integrated toolkit for performance instrumentation, measurement, and analysis for parallel, multi-threaded programs. TAU attempts to target the general computation model with different instrumentation strategies: source-code instrumentation, dynamic instrumentation, instrumented library. Its profiling and tracing currently uses PDT [163] for automatic instrumentation of C++ source code. PDT extracts the high-level interface and outputs item description to a program database. These descriptions characterize the program's functions and classes, including template instantiations, as well as template, other types, namespaces, macros, and source files.

Pablo source code instrumentation [215] supports the instrumentation with user-selected code regions. Pablo provides a GUI to support the selective instrumentation of outer loops and function calls.

AIMS [272] is a software toolkit for measurement and analysis of Fortran 77/C parallel programs based on PVM (Parallel Virtual Machine)[1]. It provides a source code instrumentor, with a GUI, to support tracing parallel programs. AIMS allows the user to specify the instrumentation of entire subroutines, communication constructs and I/O structures. AIMS allows users to select all constructs, but a subset of code regions or metrics.

A source-to-source instrumentation for C programs is introduced in [193]. C control statements, e.g. `for, while`, can be instrumented. The instrumentation is used for the optimization of an automatic reading system.

Source code instrumentation can also be conducted by compilers in which the compiler will insert instrumentation code into user program at compile time. The advantages of this approach is that the user just indicates whether or not to instrument code regions by using a small and simple set of options, and

[1] http://www.csm.ornl.gov/pvm/

the rest is carried out by the compiler. Examples of well-known tools based on compiler are gprof and CPROF. In gprof [112, 89], which is a profiling framework based on compiler, the instrumentation is done by the compiler through command line `-pg`. The CPROF system [162] is a cache performance profiler that annotates source listings to identify the source lines and data structures that cause frequent cache misses.

Recent work on an OpenMP performance interface [175] is based on directive rewriting. A general method for instrumenting OpenMP codes is presented. Directives are introduced to mark user-defined code regions and to control performance data collection. This approach does not allow the users to specify for which code regions which performance metrics should be determined. In addition, only specific OpenMP constructs, functions in the OpenMP runtime library, and user-defined code regions can be instrumented. As a result, it restricts the selection of only interesting code regions for instrumentation. This approach is similar to the use of the MPI profiling interface, which is widely used in many tools [221, 167, 179].

Most existing source instrumentation tools support limited customizable instrumentation. They mostly allow the user to select a set of code regions for which a number of pre-defined metrics are measured. Therefore, a user cannot specify the performance metrics of interest for his selected code region. Code region instrumented is restricted at level of program unit (functions, subroutines, e.g. [112, 89] ) and loop but not of arbitrary code region (some tools support the instrumentation of arbitrary code regions but manually, e.g. [221]). There is no restriction for the code region should be instrumented, e.g. the user cannot restrict few code regions in a program unit. In addition, the center of existing instrumentation techniques are code region based, not metric based. That means, given a program, the instrumentation system supports the user to select a set of code regions for which the instrumentation system will measure. However, in metric based approach, the user is interested in performance metrics collected. Thus, the user just specifies the performance metrics, e.g. communication overhead, and lets the instrumentation system perform appropriate functions, e.g. to find and instrument all code regions of which measurements may provide the user-selected performance metrics. Since the user may not have any census about which code regions may cause which performance problems, metric based approach is very helpful.

### 2.2.2 Dynamic Instrumentation for Grid Applications

*Executable code instrumentation* takes the advantage in case the source code is not available for instrumentation. In addition, the instrumentation can be done independently with programming languages and compilers. In executable code instrumentation, the instrumentation can be done before the execution of the program or during the runtime; the latter is called *dynamic instrumentation.* This approach reduces time consuming in re-compiling instrumented code of large applications due to source code instrumentation. It also allows us to instrument third party libraries and user applications whose the source code may not be available or when recompiling the source will not be possible.

PARADYN [172] is a well-known tool for automatic performance analysis of large-scale parallel programs developed in Paradyn project [9]. PARADYN applies dynamic instrumentation at the runtime of the application and searches for the bottleneck based on pre-defined constraints.

DPCL (Dynamic Probe Class Library) [75] is an object-based C++ class library that supports tool developers and users to build instrumentation tools based on dynamic instrumentation powered by Dyninst [51].

DITools [220] is an application level tool that supports dynamic interposition on dynamically-linked procedure-call boundaries.

The OMPtrace [59], MPItrace [80] and OMPItrace [81] tool instrument OpenMP, MPI, combined OpenMP and MPI parallel codes. These tools generate trace files where basic activities in a program are recorded. They support for both timing and hardware counters. OMPtrace, MPItrace and OMPItrace rely on a dynamic interception mechanism that enables them to intercept calls to the runtime library. The instrumentation code is inserted into binary file. The trace files generated by these tools can be visualized with Paraver [258].

All above-mentioned tools are based on Dyninst library [51] for manipulating binary code during or before the execution.

Etch [208] is a general-purpose tool for rewriting Win32/x86 binaries. Etch provides framework for modifying executables for both measurement and optimization. In instrumentation phase, Etch selects and instruments components of program such as instructions, basic blocks and procedures. Etch supports timing and hardware parameter measurement.

Dynamic instrumentation seems very suitable for Grid applications as the Grid is a very dynamic system whose resources can join and leave arbitrarily. Moreover, the resources on which a Grid application is executed are different from an experiment to another one, thus, source code instrumentation, in which instrumented files have to be compiled and linked on the target resources, is not well suited. However, little effort has been spent on studying and applying dynamic instrumentation techniques for the Grid. Existing tools supporting dynamic instrumentation are not designed to work with the Grid. Nor provide these tools enough accessible interfaces and interoperable protocols that can be used on the Grid. As a result, external services, e.g. performance analysis, find difficult in controlling and using them for conducting the dynamic instrumentation of Grid applications.

## 2.3 Performance Data Representation and Repository

### 2.3.1 Experiment Data Repository

Since computer systems and applications become larger and complex, a large amounts of performance data have been generated and collected over the course of performance experiments. On the one hand, to process the massive data, scalable and automatic techniques have to be developed to help the user to extract and recognize important features, and to prescribe possible solutions. Thus, performance data should be organized in a systematic way in order to enable such scalable and automatic techniques. On the other hand, performance data needs to be collected and archived for various purposes such as performance comparison, multi-experiment analysis, etc. Even though performance analysis tools proliferate, most existing tools do not organize data into a well-defined data repository; performance data is normally stored in files with specific representations.

Prophesy [235] is such a tool that employs relational database for archiving performance data. Prophesy uses performance data to perform the automatic generation of performance models.

USRA Tool family [191] collects and combines information of parallel programs from various sources at level of subroutines and loops. Information is stored in flat files which further be saved in a format understood by spreadsheet programs. APART working group [1] proposes the performance-related data specification however it does not address the implementation issues [83]. In [144], information about each experiment is represented in a Program Event and techniques for comparison between experiments are done automatically. A prototype of Program Event has been implemented in Paradyn. However, it does not provide a data repository

Previous approaches have many limitations. For example, using file repository to store performance data [191] does not provide a better infrastructure for storing, querying and exporting performance data. The lack of capability to export and share performance data has hindered external tools from using and exploiting data, e.g. in [145]. APART does not provide system-related data which is useful for correlation analysis between application- and system-specific metrics. Therefore, there is a need to develop a novel experiment data repository. Our work is stimulated by APART performance-related data model but we extend that model in several dimensions as well as we provide a robust implementation.

### 2.3.2 Semantic Representation for Performance Data

The Pablo Self-Defining Data Format (SDDF) [29] is a data description language that specifies both data record structures and data record instances. Few schemas are proposed for representing performance data such as APART [83], SCALEA [245] and Prophesy [235]. However, these approaches are tool-specific rather than widely-accepted data representations. Moreover, it is difficult to extend database schema structure to describe new resources, covering new performance metrics.

The Global Grid Forum (GGF) Network Measurements working group has created an XML data schema which provides a model for network measurement data [106], but not for applications. Similarly, GLUE schema [237] defines a conceptual data model to describe computing and storage elements and networks. In [73], ontology has been applied for improving the semantic expressiveness of network management information and the integration of information definitions specified by different network managements. None of these schemas models concepts of application experiments.

Our approach to describe the semantics of performance data is based on ontology by using OWL (Web Ontology Language) [188]; OWL can express data in a high semantic level. The data models in the above-mentioned approaches do not support knowledge discovery via inference whereas the ontology model does. Therefore, building a reasoner on that models is more intensive work, difficult and costly. Different from the above-mentioned schemas, our ontology focuses on representing performance data of both applications and resources. The modeled objects in GGF and GLUE schema are quite generic for resource objects, e.g. networks and computational nodes, thus vocabularies and terminologies of these schemas can be incorporated into our resource-related ontology.

CIM [66] is a very generic model for describing overall management information in a network/enterprise environment. Our work is different as we are focusing on describing performance data using existing models such as OWL.

Recent work in [233] describes how ontology can be used for matching resource in the Grid. That confirms that an ontology for describing performance data of both computational resources and applications can be a useful tool which can be used to provide performance data for matching resources in the Grid.

## 2.4 Performance Analysis for Parallel Programs

Over the past few years, numerous performance analysis tools have been developed to assist application developers to analyze the performance of their applications on SMP clusters. These tools fall into two categories: tracing analyzers, e.g. [272, 204, 279, 179, 221, 268, 196] and profiling analyzers, e.g. [260, 221, 6, 170]. In this section, we discuss our related work with respect to distinct features of our analyses.

### 2.4.1 Overhead Analysis for Parallel Programs

Overhead analysis is a powerful conceptual tool in understanding the performance behavior of parallel programs. Overheads associated with parallelism are classified into temporal and spatial overheads. Temporal overhead has an impact on the execution time of a program whereas spatial overhead influences demands on memory.

Temporal overhead, defined based on Amdahl's law [25], is the difference between achieved and optimal parallelization. It is important to determine which factors cause the temporal overhead in which part of the code. Based on that, further tuning techniques can be applied in order to improve the performance of parallel programs. Overhead analysis is powerful because it not only provides the programmer insights about the performance characteristics of programs but also reveals why the programs behave poorly.

The approach of using overhead analysis for performance analysis is not new. Crovella et al. called temporal overhead as *lost cycles* [68]. They categorized overhead into *Load Imbalance*, *Insufficient Parallelism*, *Synchronization Loss*, *Communication Loss*, and *Resource Contention*. Bull then proposed a classification of temporal and spatial overheads [52]. Bull's classification of temporal overheads is designed to meet criteria proposed by Crovella et al. Temporal overheads are classified into: *Information Movement*, *Critical Path*, *Control of Parallelism*, *Additional Overhead*, and *Unidentified Overhead*.

FINESSE [178] is a prototype environment designed to support rapid development of parallel programs for single-address-pace computers. Performance is interpreted in terms of the extra execution time associated with a small number of categories of parallel overhead including *Version cost, Unparallelized code cost, Load imbalance cost, Memory access cost, Scheduling cost, Unexplained cost.* FINESSE determines improvements in performance relative to either a base reference implementation or a previous version in the performance improvement history.

OVALTINE [35] is a tool that automates the determination of the overheads of a given OpenMP implementation with respect to a given sequential implementation. Currently OVALTINE supports only Fortran 77 source files. OVALTINE uses the classification described in [52] for the overhead analysis.

A speedup component model for shared-memory parallel programs is presented in [149]. The speedup component model categorizes overhead factors into several main components: memory stalls, code overhead, thread management, and computation overhead.

Previous overhead classifications [52, 68] do not cover various ranges of programming paradigms such as OpenMP, MPI and mixed parallel programs. They do not exploit overheads specifically bound in these parallel programs. There is a lack of connection between overhead classification and instrumentation in these tools. They normally treat the instrumentation as a different aspect. In performance analysis phase, these tools analyze performance overheads, however, the performance analysis of these tools does not instruct their instrumentation systems to collect and measure only performance overheads of interest for selected code regions.

### 2.4.2 Search and Filter Performance Data

Whereas there exist performance tools that provide bottleneck search capabilities [55, 86], another issue is that most existing performance tools lack basic search and filter capabilities. Commonly, the tools allow the user to browse code regions and associated performance metrics through various views of performance data. For example, tools such as [279, 109] provide numerous displays including process time-lines with zooming and scrolling, histograms of state durations and message data. Those views are crucial but mostly require all data to be loaded into the memory, eventually making the tools in-scalable. Moreover, the user has difficulty in finding out the occurrence of events with interesting criteria of performance metrics, e.g. code regions with overhead of data movement [244] larger than 50% of total execution time. Search of performance data will allow the user to quickly identify interesting code regions. Filtering performance data being visualized helps to increase the scalability of performance tools. Utilizing a data repository allows us to power the archive, to facilitate search and filter with great flexibility and robustness based on SQL language, and to minimize the implementation cost.

### 2.4.3 Multi-Experiment Analysis

Most performance tools investigate the performance for individual experiments one at a time, e.g. [172, 167, 179, 204, 258, 82].

Karavanic et al. presented techniques for the quantitative comparison of several experiments, and performance diagnosis based on dynamic instrumentation [144]. Performance analysis is done automatically for every experiment based on historical data [143].

Recently, Song et al. presented an algebra which can be used to compare, integrate, and summarize performance data from multiple sources [226]. The algebra supports arithmetic operations to merge, subtract and average performance data from different experiments.

### 2.4.4 Soft Performance Analysis

Soft computing has been widely applied in industrial electronics, automotive systems and manufacturing, intelligent robotic systems, decision-support systems, system engineering monitoring, etc[2]. Fuzzy logic has recently been used in performance monitoring of parallel and distributed program, e.g. to make sure applications running under performance contracts [263], adaptive control of distributed application [207]. However, soft computing, especially fuzzy logic, has not been exploited in data analysis techniques, e.g., performance classification, performance search, of existing performance analysis tools for parallel and distributed programs.

We propose and develop a set of techniques for performance analysis of parallel programs; the techniques are based on soft computing. In our approach, we use fuzzy logic to develop a concept of performance score and we introduce a concept of performance similarity based on similarity theory. Several soft techniques for performance analysis are proposed and developed,

[2] see series *Studies in Fuzziness and Soft Computing* published by Springer (http://www.springeronline.com).

such as fuzzy-based classification, fuzzy-based performance search, similarity analysis, fuzzy clustering analysis and fuzzy rules for reducing data.

The APART working group introduces the concept of performance property [83] that characterizes a specific negative performance behavior of code regions. However, performance property is associated with a single performance metric. A performance property cannot represent a set of performance metrics. Instead, performance properties can be grouped into a collection, as discussed in [87]. However, there is no concept of weight operator associated with performance properties. A set of performance properties is just a collection with simple operators, e.g. min and max. Our performance score is based on theory of fuzzy logic. Fuzzy logic also allows the representation of fuzzy concepts, such as *near* and *very*, which cannot be found in performance properties. In addition, performance score can be computed based on linear and non-linear model with various membership functions where performance property is computed based on linear model (mostly employ ratio-based schema).

Similarity of profiling data has been used in guiding adaptive compilation [150]. However, it applies only profiling data in a run that aims to provide information for optimizing code in compiler. In [56] dispersion statistics is used to characterize the load imbalance by measuring the dissimilarity of performance metrics. Performance metrics are normalized by measuring deviation from a mean value of a data set. Dissimilarity is characterized based on Euclidean distance of normalized metrics. Our similarity measure is based on normalized metrics which are computed based on fuzzy logic and normalized into the range $[0, 1]$. Our similarity measure can be applied for code regions (e.g. load imbalance) and for experiment factors.

Recent work has focused on developing scalable analysis methods for performance analysis. Vetter presented classification based on machine learning that has been used for classifying performance characteristics of communication in parallel programs [261]. Ahl and Vetter used multivariate statistical techniques on hardware performance metrics to characterize the system [22]. They also evaluated the use of clustering and factor analysis to extract performance information. However, only joining (tree) and k-means clustering are examined. They do not deal cases in which multiple variables with different scales and weight factors. In [211], statistical analysis is used to study different factors that affect the mapping process of scientific computing algorithms to advanced architectures. The use of design of experiments, analysis of variance (ANOVA) correlation and subset feature selection has applied to performance data to reveal the relationship between high-level abstracts to low-level performance information.

In TAU, simple rules have been used to reduce the volume of tracing data [166]. The rules support the specification of simple conditions in the form of crisp expressions of performance metrics that can exclude some important data. As the user usually has no priori knowledge about the experiment, soft conditions for filtering data are more suitable.

Existing techniques are based on crisp and numerical analysis, employing some machine learning techniques, but not fuzzy logic and the combination of the two. They do not consider cases in which multiple variables with different scales and weight factors. Our work differs from existing techniques as we use fuzzy logic to represent performance data with different scales and different weight factors. We focus on exploiting soft computing techniques, e.g. fuzzy logic, machine learning and the combination of the two, to provide soft techniques to classify, search and analyze performance data. Our work complements existing scalable and intelligent methods for performance analysis by introducing a new set of fuzzy-based analysis techniques to the performance analysis of parallel and distributed programs.

## 2.5 Performance Monitoring and Data Integration Middleware for the Grid

Recently, work that exploits peer-to-peer (P2P) features for Grid computing has shown many advantages, e.g. in [129, 232, 133]. However, P2P techniques have not been exploited in Grid monitoring middleware.

Over the past few years, numerous Grid monitoring and performance tools have been developed, as studied in [104]. Several existing tools are available for monitoring Grid computing resources and networks, e.g. MDS (also known as GRIS) [70], NWS [269], GridRM [31], Gangila [168], GDMonitoring [65].

Grid Monitoring Architecture (GMA) [30] proposal of Global Grid Forum (GGF) [107] describes the major components of a Grid monitoring architecture and their essential interactions. GMA provides standard terminology and describes a specification to support the development of interoperable monitoring tools for the Grid. The DataGrid[3] has developed a Relational Grid Monitoring Architecture (R-GMA) [19] for distributed resources. R-GMA exposes a relational database model with SQL support to provide static as well as dynamic information about Grid resources.

Network Weather Service (NWS) [269] is a generalizable and extensible facility designed to provide resource performance forecasts in metacomputing environments. NWS provides a set of system sensors, such as CPU and network sensors, that periodically monitors and dynamically forecasts the performance of various networks and computational resources.

JAMM Monitoring System [241] is an automated agent-based architecture to support monitoring data about computer systems. JAMM provides sensors wrapping common utilities such as *netstat*, *iostat* and *vmstat*. JAMM lacks support for application monitoring and performance analysis.

Ganglia [168] is a monitoring system targeting to wide area cluster. Ganglia is comprised of two components: Gmon for local-area monitoring and Gmeta for wide-area system. At the cluster level, Gmon uses UDP multicast protocol

[3] http://eu-datagrid.web.cern.ch/eu-datagrid/

to monitor state within a single cluster. Gmons are organized into hierarchical monitoring tree by using Gmetas. A Gmon communicates with its Gmeta by using XML streams atop TCP connections. Ganglia stores monitoring data in time-series databases based on RRD (Round Robin Database) [3]. With Ganglia, however, it is difficult to construct the tree hierarchy because Gmeta needs a priori knowledge of other nodes. Ganglia supports a large set of predefined metrics. However, it is not quite sure how monitoring data of (Grid) applications/services can be published into the middleware.

Mark Baker and Garry Smith present a prototype Grid-site Monitoring System (GridRM) [31] which is based on three-tier architecture. At each grid site, a GridRM Gateway is employed to control access to local resource information and to provide a mechanism for the user to query the status of remote resources. GridRM Gateway provides a web client for the user to control and conduct the monitoring. GridRM Gateway actually accesses information of resources from other services such as MDS and SNMP. At each Gateway, a relational database is employed to store resource monitoring information and general site metadata.

## 2.6 Performance Analysis of Grid Workflow-based Applications

NetLogger [240, 118] is a distributed application, host and network logger. However, application sensors have to be instrumented manually. NetLogger supports only post-mortem analysis of applications based on event traces.

GRM [196] is a semi-on-line monitor that collects information about an application running in a distributed heterogeneous system and delivers the collected information to the PROVE visualization. The application is manually instrumented by inserting C functions. On the host where the application processes are running, a local monitor is used to handle trace data recorded by application sensors and stored in shared memory buffer. The local monitor will send trace data to the main monitor when the shared memory buffer is full. PROVE conducts off-line or semi-online analysis based on data provided by the centralized main monitor of GRM.

Autopilot [207] is a distributed performance measurement and resource control system that is based on the Pablo toolkit [204]. Autopilot's sensors are embedded in a program prior to program execution to collect and measure performance data. Autopilot's actuators are used to enable and configure application behavior and resource management policies. An Autopilot Manager is used to maintain information about available sensors and actuators, and their associated properties. Autopilot also captures various system performance statistics on a set of machines. Performance data is stored in Self-Defining Data Format (SDDF) files.

OCM-G [34] is an infrastructure for Grid application monitoring which uses OMIS [165] as a communication interface. Application monitor embedded in the monitored application collects performance data and passes the data to a local monitor which resides on each node. A service manager is used to control several local monitors. One of service managers named main service manager is used to hold collected information about the whole application.

G-PM [50] is a performance analysis tool targeting to interactive Grid application. G-PM allows the user to conduct the performance analysis online during the execution of the application. However, application code is instrumented only by linking with wrapper libraries consisting of probes used to collect performance data. Instrumentation code can be activated or deactivated by controlling conditional variables. G-PM uses OCM-G as its based Grid monitoring system.

None of aforementioned systems provides OGSA-capable services. They support monitoring and performance analysis of scientific applications, but workflow-based applications, executed on the Grid.

Monitoring of workflows is an indispensable part of any WfMS (Workflow Management System). Therefore it has been discussed for many years. Many techniques have been introduced to study quality of service and performance models of workflows, e.g. [148, 58], and to support monitoring and analysis of the execution of the workflow on distributed systems, e.g. in [212, 21]. Our work and existing work share many general concepts of performance metrics of and monitoring techniques for the workflow in distributed systems. However, existing work concentrates on business workflows and Web services processes. Little effort has been spent on studying performance metrics and on monitoring and analyzing the performance of scientific workflows executed on Grids which are more diverse, dynamic and inter-organizational.

Most effort on supporting the scientist to develop Grid workflow-based applications is focused on workflow language, workflow construction and execution systems[4], but not concentrated on monitoring and performance analysis of the Grid workflows. P-GRADE [142] is one of few tools that supports tracing of workflow applications. Instrumentation probes are automatically generated from the graphical representation of the application. It, however, limits to MPI and PVM applications.

## 2.7 Summary

Classification of temporal overhead and techniques to explain performance problems for shared and distributed memory parallel programs have not been fully investigated for new programming paradigms such as hybrid parallel OpenMP/MPI. Creating a performance tool that provides customizable instrumentation and measurement system, which can be precisely controlled by user to specify code regions of which performance metrics should be determined, has not been investigated. Most of current tools lack conceptual and

[4] e.g., see Scientific Workflows Survey at http://www.extreme.indiana.edu/swf-survey/

systematic organization for performance-related data representation. In addition, performance data sharing and tool integration have not been focused. The performance analysis still is a daunting task because of the lack of support of performance search and multiple experiments analysis.

Current techniques in existing performance analysis tools have mainly been used to process the performance data that are in the form of precise numerical and measurement-based data. These techniques apply *exact* analysis methods that result in exact conclusions about performance characteristics of applications. Moreover, performance tools based on numerical analysis interact with the user through complex numerical values which are not easily understood by the user.

Most existing Grid monitoring tools are separated into two distinct domains based on what they are monitoring: *Grid infrastructure monitoring* and *Grid application monitoring*. The lack of combination of two domains in a single system has hindered the user from correlating measurement metrics of various sources at different levels when conducting the monitoring and performance analysis. In addition, most existing Grid monitoring tools focus on the monitoring and analysis for Grid infrastructure; yet little effort has been done for integrating and analyzing performance data of Grid applications.

Although existing Grid monitoring tools have monitoring sensors operating in a distributed manner, these tools do not focus on the interoperability among sensor networks and the self-organization within them, and support limited types of sensors. Mostly they support only event-driven sensors. Sensor managers are configured into tree of point-to-point connections; or directory services, supporting discovery of data and sensor managers, do not interact with each other. Currently data discovery and DQS (data query and subscription) are mostly based on hierarchical or centralized models. However such models do not work well with more dynamic, large-scale distributed environments in which the structure of resources frequently changes.

To date, most existing Grid performance analysis tools are not based on Grid service-oriented model because most of them are originally designed for and targeted to conventional parallel and distributed systems (e.g. clusters, SMP machines). As a result, these performance analysis tools do not well address challenges in Grid environments such as scalability, online analysis and security. There is a lack of suitable tools that support dynamic instrumentation for Grid applications and performance analysis of Grid workflow-based applications.

# 3

# Model

## 3.1 Introduction

Before we begin to expose our techniques and methods in detail, we present a few preliminaries. First, we describe parallel architecture models and the programming paradigms on these models. Second, we describe the Grid computing and the workflow-based applications executed on the Grid.

We present the concept of code regions and the classification of code regions. We introduce a novel representation for code regions named dynamic code region call graph (DRG). The DRG reflects the dynamic relationship between code regions and its subregions and enables a detailed overhead analysis for every code region. We outline performance experiment model and performance metrics used in the rest of this dissertation.

## 3.2 Parallel Architecture Models

Traditionally, computer organizations are categorized based on Flynn's Taxonomy [91]. Flynn's classification is based on programming models used on different machines and includes four categories named SISD, SIMD, MISD and MIMD.

- *SISD (Single Instruction, Single Data)*: In this type, machines have one CPU and thus can only execute one instruction stream serially.
- *SIMD (Single Instruction, Multiple Data)*: In this type, machines have a large number of identical processors. All the processors simultaneously execute same instruction, in lock-step, on distinct data streams.
- *MISD (Multiple Instructions, Single Data)*: In this type, machines theoretically execute multiple instructions on a single data. However, no machines of this type have been built.
- *MIMD (Multiple Instructions, Multiple Data)*: In this type, machines execute several instruction streams in parallel on different data streams.

However, presently parallel systems are normally classified into shared memory systems and distributed memory systems based on the memory organization of these systems [259].

- *Shared Memory Systems*: have multiple CPUs which share the same address space. Shared memory systems can be SIMD or MIMD. Examples of this type are SMP (Symmetric Multi-Processor), PVP (Parallel Vector Processing).
- *Distributed Memory Systems*: have multiple CPUs and each CPU has its own associated memory. The CPUs are connected by interconnecting networks. Distributed memory systems can be SIMD or MIMD. Examples of this type are MPP (Massively Parallel Processing) and Clusters.

In the following we briefly discuss SMP, MPP and cluster model, which are the architectures that we support. More detail of parallel machines can be found in [69, 194, 186, 53, 259, 131, 156, 198].

### 3.2.1 Symmetric Multi-Processor Machine

Symmetric Multi-Processor (SMP) machines [69] are characterized by tightly-coupled series of identical processors that share common memory modules. As depicted in Figure 3.1, in the SMP architecture, all processors and memory modules are linked through an interconnect, typically a bus, crossbar, or multistage interconnect [69]. In practice, each processor of SMP machine may have its own local cache or the shared-memory has the hierarchical form. In the SMP architecture, memory access time is the same on all processors.

As each processor can directly access to all memory, the SMP architecture directly supports the globally shared address space programming model. Processes exchange data and coordinate with each other through reading and writing the shared address space. Message passing programming model is also supported on SMP architecture but through the use of system software that implements send/receive operations on top of read/write operations.

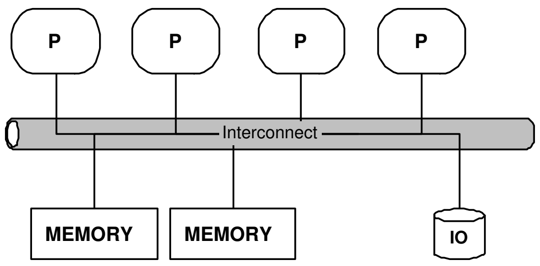


**Fig. 3.1.** SMP machine model.

The SMP architecture is increasingly widely used for a wide range of machines, from mid-range machines with few processors to supercomputers with hundreds of processors[1]. One of the main reasons is that while SMP provides high throughput and performance through multiprocessing, it is relatively straightforward to develop parallel programs on the SMP architecture. Further information about SMP machines can be found in [69, 259].

### 3.2.2 Massively Parallel Processing Machine

Massively Parallel Processing (MPP) machine [198], also known as multicomputer [96], has a distributed-memory multiprocessor architecture. Figure 3.2 illustrates MPP machine model. MPP machine comprises a number of nodes, each node is a computer containing a CPU and memory; nodes are linked by an interconnect. The structures of interconnecting network of MPP normally employ hypercube, flat tree, or 2-D/3-D mesh topology [259].

In MPP model, a processor of a node cannot directly access to memory in another node. Nodes communicate with each other by sending and receiving messages over the interconnect. In the ideal interconnect, communication time between two nodes is independent of node locations. Obviously, memory access time on local memory (in the same node) is much smaller than that on non-local memory (in different node).

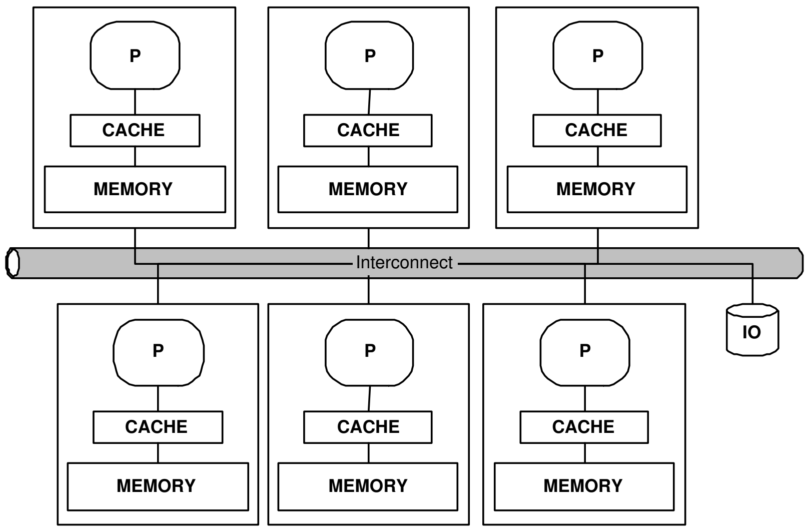


**Fig. 3.2.** MPP machine model.

[1] The number of processors of most SMP machines is smaller than 100. An example of SMP machines with more than 100 processors is Fujitsu PRIMEPOWER 2500 whose maximum number of processors is 128 (see http://www.fujitsu.com/global/services/computing/server/unix/enterprise/pw2500).

Due to their loosely couple model, MPP systems normally have more processors than SMP ones. This makes MPP systems become the choice for large-scale parallel computers. Further information about MPP machines can be found in [198, 259].

### 3.2.3 Cluster Model

Clusters offer modest parallel computing systems that can be built using commodity computers, e.g. workstations and PCs (Personal Computers), and local area network (LAN) technology [194, 186, 53]. Gregory F. Pfister defines a cluster as follows.

**Definition 3.1 (Cluster).** *"A cluster is a type of parallel or distributed system that (i) consists of a collection of interconnected whole computers, (ii) and is utilized as a single, unified computing resource". [194]*

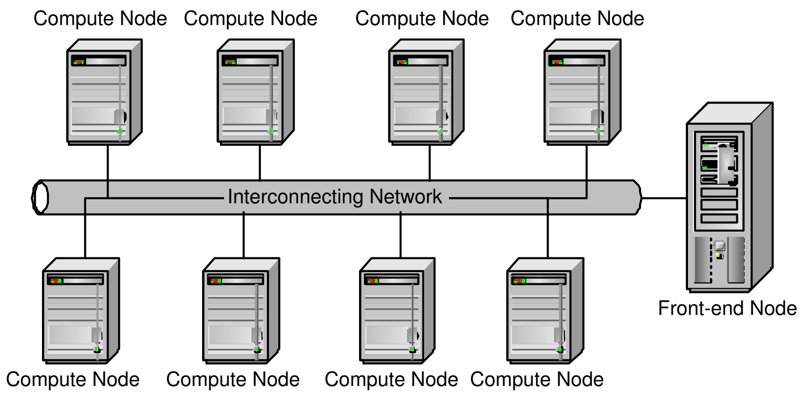


**Fig. 3.3.** Model of a cluster machine.

The cluster, illustrated in Figure 3.3, is a distributed memory parallel system. Normally, each cluster provides a special node called the front-end node. The user uses the front-end node to edit programs, to compile code, to submit jobs, etc. The outsider views and utilizes the cluster as a single computing resource. The cluster model requires additional system software to aggregate network of compute nodes onto a virtual unified parallel computer; these software basically provide (1) utilities for initializing and starting parallel applications on nodes of the cluster, and (2) a library (e.g. message passing system) used by parallel applications to exchange data between cluster nodes [194, 259].

Cluster and MPP systems are similar as both have distributed memory architecture. However, they differ in (i) the interconnect between processors and (ii) architecture of nodes. The interconnect in MPP systems is high bandwidth and low latency while the interconnecting network in clusters normally is based on commodity LAN technologies. Both MPP and cluster systems normally have off-the-shelf processors. However, usually not all MPP nodes have complete I/O resources and MPP nodes normally do not have complete conventional operating system facilities, but a specific customized operating system. Cluster nodes normally have a completed and conventional operating system.

An SMP cluster is a special type of cluster systems in which nodes are SMP machines. The architecture of SMP cluster fosters the hybrid parallel programming models that expose both message passing and shared memory models. The emerging programming trend on SMP clusters is to combine message passing and shared memory. For instance, MPI is used for communication between SMP nodes, and multi-thread or OpenMP is used between processors within an SMP node.

More detail of cluster systems can be found in [194, 186, 53, 259].

## 3.3 Parallel Programming Models

Our framework supports a variety of parallel programming models including OpenMP, MPI, HPF+ and hybrid parallel models including OpenMP/MPI and HPF+/OpenMP. In this section, we briefly describe these programming models. Further detail of these parallel programming models can be found in [266, 61, 225, 96, 115, 127].

### 3.3.1 Shared Memory with OpenMP

The shared memory programming model exploits the parallelism in shared memory multi-processor in a portable way. The most well-known standard established by the industry is OpenMP (Open Multi-Processing) [266, 61]. OpenMP is a parallel programming model for shared memory and distributed shared memory on multi-processor. OpenMP works in conjunction with existing languages, such as C/C++, Fortran and Java, by introducing a set of directives for specifying the shared memory parallelism in source codes, a small set of runtime library routines, and environment variables.

OpenMP uses the fork-join model of parallel execution [47, 61]. Figure 3.4 shows the execution model of OpenMP programs described in the OpenMP specification [47]. An OpenMP program begins execution as a single-threaded process, referred to as the master thread. The master thread executes sequentially until it encounters a parallel construct. Before executing the parallel construct, additional threads will be created and all threads form a team of threads with the master thread as the master of the team. Threads in the team execute, in parallel, all program statements enclosed by the parallel construct. At the end of the execution of the parallel construct, the threads in a team synchronize and only the master thread continues the execution.

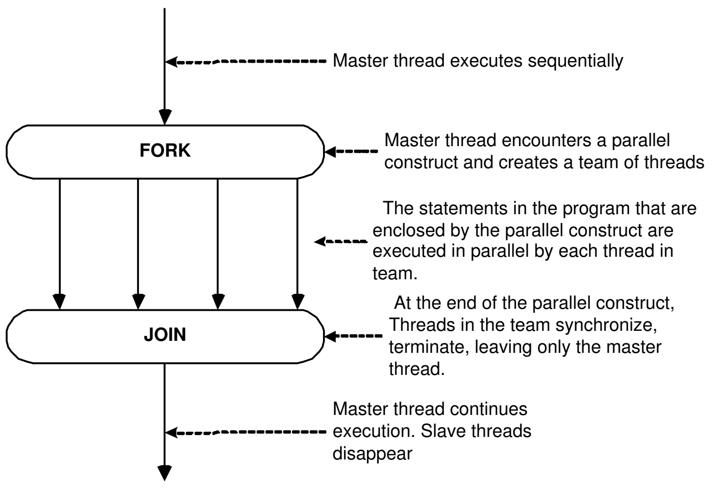

Master thread executes sequentially
FORK
Master thread encounters a parallel construct and creates a team of threads
The statements in the program that are enclosed by the parallel construct are executed in parallel by each thread in team.
JOIN
At the end of the parallel construct, Threads in the team synchronize, terminate, leaving only the master thread.
Master thread continues execution. Slave threads disappear

**Fig. 3.4.** Execution model of OpenMP programs described in [47].

### 3.3.2 Message Passing with MPI

In the message passing model [225, 96, 115], each process has a private address space that cannot be accessed by other processes. A process has to communicate explicitly with other processes in order to exchange the data and to coordinate the work. Communication is performed through send/receive operations and both sender and receiver processes are involved explicitly. The sender has to know and specify to whom it sends data. Normally, data is packed at the sender side and unpacked at receiver side because the data is exchanged on heterogeneous systems.

The *Message Passing Interface* (MPI) [225, 115, 124] currently is the most widely supported and used standard for programming of distributed memory parallel systems in message passing models. MPI is also currently used for parallel programming on shared memory systems and hybrid shared/distributed memory systems. We mostly use MPICH [114], a portable implementation of MPI, in our performance experiments. MPICH supports message passing via sockets or shared memory. Other MPI implementations, e.g. LAM [7], are also working well with our tools.

### 3.3.3 Hybrid Parallel Paradigm with OpenMP/MPI

Figure 3.5 shows the execution model of a generic process in a hybrid parallel program. The process executes a program which is subdivided into sequential and parallel regions. A process may dynamically spawn/fork, synchronize, and terminate threads during execution of the program. All threads of the process share the same address space. In a sequential region only one thread within the process is active (executes the code region). In a parallel region several threads may be active and execute simultaneously. Threads may be spawned at the beginning of the program or at the beginning of the parallel region which is language/implementation dependent. At the end of the parallel region the active threads may be synchronized (e.g. through a barrier synchronization or join operation), and only one thread continues to execute the following region; other threads will either be terminated or turned passive (do not execute a region but remain alive) at the end of a parallel region. A passive thread can be turned active in a subsequent parallel region or terminated at the end of the program execution. Active threads within the same process exchange data through the shared memory. Exchanging data between different processes is enabled only between active threads, associated with different processes, through generic send and receive operations which can be executed in both sequential and parallel code regions.

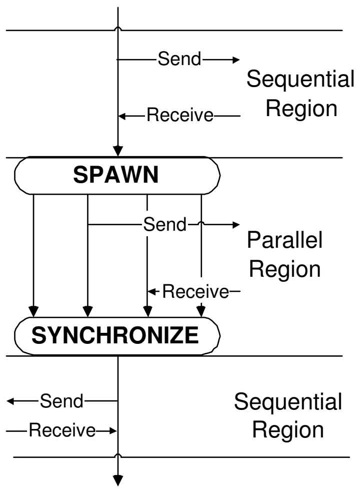


**Fig. 3.5.** Execution model of a single process in hybrid parallel program.

We support performance analysis for hybrid parallel programs based on a mixed execution model comprising both distributed and shared memory parallelism. Our current implementation uses mostly OpenMP as the shared memory programming language and MPI for message passing.

### 3.3.4 Data Parallelism and Hybrid Data Parallelism

Data parallelism exploits the concurrency that applies the same operation to multiple elements of a data. The concept of data parallel programming originates from SIMD style of programming. The most well-known language that supports data parallelism features is High Performance Fortran (HPF).

HPF [127] is a superset of Fortran 90. In HPF the compiler normally handles the parallelism. The programmer puts directives specifying the data distribution for the tasks into the source code. The compiler then conducts the parallelization. Because the HPF compiler hides all the detail of the parallelism, in order to capture performance information about parallelized code, the performance tools have to work closely with the compiler.

HPF+ [40, 39] is an improved version of HPF that provides new features for an efficient handling of irregular codes. HPF/OpenMP [38] is an extension of HPF for clusters of SMPs. HPF/OpenMP provides a mechanism for specifying the hierarchical structure of SMP clusters in HPF programs, mapping data arrays onto a set of SMP nodes and within an SMP node. Both HPF+ and HPF/OpenMP have been implemented in VFC compiler [37].

Because the HPF compiler hides all parallel tasks, automatically generated from high level specifications provided by the developer, from the developer, the performance of an HPF program depends largely on the capabilities of the compiler. Therefore, the performance tool should be able to capture performance metrics that characterize HPF programs such as sequential bottleneck, communication costs, etc., by measuring generated HPF code.

We support instrumentation, measurement and performance analysis of data parallelism programs developed with VFC compiler. In VFC complier, HPF+ [39] , HPF+/OpenMP code will be translated into Fortran90/MPI and Fortran90 MPI/OpenMP, respectively. Thus, the execution model of HPF+ and HPF+/OpenMP follow the execution model of MPI and OpenMP/MPI, respectively, as discussed in Section 3.3.2 and 3.3.3.

## 3.4 The Grid

The term *the Grid* [99, 41] was initially used to indicate *"Computational Grid"*. Ian Foster and colleagues define Computational Grid as follows.

**Definition 3.2 (Computational Grid).** *"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities". [99]*

However, over the past few years, the Grid has been developed and extended substantially and Grid computing has been applied to a wide range of domains. As a result, the current Grid is much differently from the meaning of its initial definition [147]. The basic tenet of the Grid concept is *"coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations"* [147]. In an article on ComputerWorld[2], Sami Lais defines Grid computing as follows

[2] www.computerword.com

**Definition 3.3 (Grid computing).** *"Grid computing is a hardware and software infrastructure that clusters and integrates high-end computers, networks, databases and scientific instruments from multiple sources to form a virtual supercomputer on which users can work collaboratively". [157]*

As stated in [97], a Grid system, which is different from conventional distributed systems, must have the following properties:

- coordinating resources that are not dependent on and are not controlled by a centralized authority
- utilizing and providing protocols and interfaces that are standard, suitable to be used for multiple basic purposes.
- delivering numerous and diverse qualities of service

The Grid computing is commonly illustrated by analogy with a power grid [99, 199, 63]. When we connect the power plug of a device into an electrical socket, we expect that the device gets the correct electric power without concern about the actual source of that power. Similarly, a Grid infrastructure consists of many diverse resources, e.g. computers, networks, database. A user has access to the Grid in order to get the computing power that the user needs. However, individual resources of the Grid will not be visible to the user and the user is not necessarily aware of how the resources are gathered and assembled. To make this vision become reality, Grid computing must be based on standards that address the interoperability and integration of that many diverse resources. Currently, the Open Grid Services Architecture (OGSA) [93, 94] and the Globus Toolkit [108] play a major role in providing the necessary framework for building the above-mentioned vision.

### 3.4.1 Types of Grids

Grids can be categorized based on the type of applications that they support (e.g, in [113, 135]) or based on the organization of the Grid infrastructure (e.g., in [238, 203, 141]). As presented in [113, 135], with respect to the applications that the Grid supports, three primary types of Grids are computational grid, scavenging grid, and data grid.

- *Computational grid*: A computational grid [99] provides high-end computational capabilities. Most computational nodes in computational grid are high-end computational machines (e.g. SMP, MPP, Cluster).
- *Scavenging grid*: A scavenging grid commonly harnesses the available computing power of a large number of personal computers which are usually available as resources of the Grid when they are not used by their owners.
- *Data grid*: A data grid [62] is a specialization and extension of the Grid that aims at providing an integrated infrastructure for housing, accessing and utilizing large data collections across multiple organizations.

The peer-to-peer computing model [173] is also considered as another form of Grid computing [98, 17, 122]. While we do not support the performance monitoring and analysis of peer-to-peer systems, we exploit ideas and features of peer-to-peer computing on the design and development of a middleware for performance monitoring and data integration in the Grid.

### 3.4.2 Grid Computing Environment Model

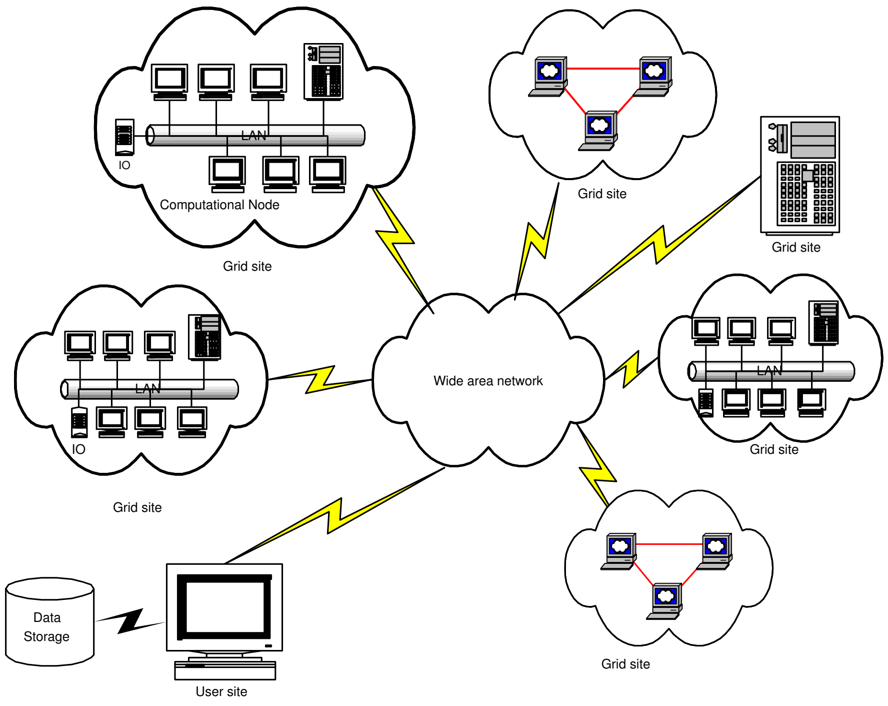


**Fig. 3.6.** Grid environment model.

Our Grid environment is mainly computational grid, but also mixed with scavenging grid. Figure 3.6 presents the Grid computing environment model on which we will conduct the performance monitoring and analysis. A Grid environment is viewed as a set of Grid sites.

**Definition 3.4 (Grid site).** *A Grid site is comprised of a set of grid services, within a single organization that is utilized as a single, unified computing service.*

A Grid site consists of a number of *computational nodes* (or hosts, computers) that share the same security infrastructure. Computational nodes in a Grid site exchange data with each other through a local network and are utilized as a single, unified computing resource. Computational nodes in a Grid site are controlled by a single resource management service. A computational node can be any computing platform, from a single processor workstation to an SMP to an MPP system. A Grid site may contain only a single computational node or may consist of a cluster.

Each computational node may have single or multiple *processor(s)*. On each computational node, multiple *application processes* execute, each process may have multiple *threads* of execution.

Grid sites exchange data with each other through a, mostly wide area, network. The user may reside in one of Grid sites or in a different site. Depending on a particular configuration of a Grid site, a computational node of that Grid site may or may not directly communicate with another computational node within another Grid site.

With respect to the organization of a Grid, our *Grid site* is similar to *IntraGrid* [203, 141], InfraGrid [141] or *Cluster Grid* [238].

### 3.4.3 Open Grid Services Architecture

The evolution of the Grid has been a continuous process. The current generation of the Grid [209] follows the service-oriented approach [190, 132, 154, 264]. A service-oriented Grid is an important move because the service-oriented approach aims to achieve the *interoperability* and *integration*, which are key issues of the Grid, among distributed applications. The Open Grid Services Architecture (OGSA) represents an evolution towards a Grid system architecture that is based on Web services concepts and technologies [154, 264]. In OGSA, the Grid is considered as a set of Grid services. A Grid service in OGSA is defined as follows.

**Definition 3.5 (Grid service).** *"A Grid service is a Web service that provides a set of well-defined interfaces that follows specific conventions. The interfaces address discovery, dynamic service creation, lifetime management, notification, and manageability; the conventions address naming and upgradability". [94]*

In OGSA [94], the interfaces, or *portTypes* in WSDL (Web Service Description Language) terms [14], address:

- *Discovery*: provides mechanisms for discovering available services and determining the characteristics of those services.
- *Dynamic service creation*: provides mechanisms for dynamically creating and managing new service instances. A new service instance can be created by a *factory service*.
- *Lifetime management*: provides mechanisms for handling the fault tolerance of services.
- *Notification*: provides mechanisms for services to notify each others asynchronously of interesting changes of state.

- *Manageability*: addresses authorization, policy management, monitoring and management of Grid service instances.

The conventions address version management and compatibility between services. Details of OGSA can be found in [94].

As the Grid is moving towards service-oriented architecture, and with the proliferation of service-oriented Grid middleware and applications, monitoring and performance analysis tools for the Grid should be Grid service-based as well. Grid performance analysis tools based on Grid service not only can smoothly access and monitor Grid services and be easily deployed in the Grid environments but also are easily being integrated in and interoperated with other Grid tools/services. Our Grid performance analysis framework is OGSA-based.

### 3.4.4 Globus Toolkit

The current de facto standard middleware for the Grid is the Globus Toolkit (GT) [108]. GT provides fundamental services to securely access and to manage distributed shared resources. Three main services of the GT are:

- Grid Security Infrastructure (GSI): provides security for the Grid that is based on public key infrastructure.
- Grid Resource Management (GRAM): provides mechanism to submit and execute jobs onto remote resources.
- Grid Information Service (GIS), also known as MDS (Monitoring and Discovery System): provides facilities for discovering the available resources.

## 3.5 Grid Workflow-based Applications

Grid programmers face many difficulties in developing Grid applications due to the very dynamic and diverse nature of the Grid. Grid programmers have to manage the computation in *"an open-ended, heterogeneous, and dynamic"* environment and to design *"the interaction between remote services, data sources, and hardware resources"* [54]. Moreover, Grid programmers suffer from the lack of high-level development tools, e.g., performance monitoring and analysis.

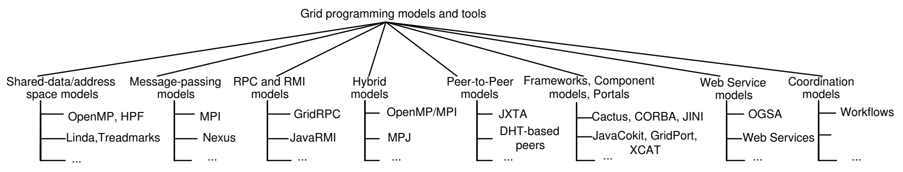


**Fig. 3.7.** Grid programming models and tools (based on [54]).

Figure 3.7 presents current programming models used in the Grid, as surveyed in [54]. While existing Grid applications are built by using the current programming tools, yet the open question is which programming models suited for the Grid because it seems that the current tools and languages are insufficient to support the effective development of Grid applications [54]. This situation requires strong research efforts in the design of programming models and languages supporting the development of Grid applications.

Meanwhile, there is a growing trend on using workflow-based applications on the Grid [164, 155, 236, 74]. Workflow applications currently represent the most interesting class of Grid applications that are truly distributed: using multiple Grid sites for a single application execution at the same time. The workflow, on the one hand, exploits the coarse-grained computing model on which activities (tasks), executed on different resources in Grid sites, can interact together in order to solve a particular problem. On the other hand, an activity of a workflow can exploit various fine-grained parallelism, by using OpenMP, MPI, etc., on a Grid site. Grid workflow applications raise a lot of interest in the Grid community to further increase the class of potential applications for the Grid. However, the Grid workflow community suffers from the lack of the supportive tools, e.g. performance monitoring and analysis, and of the workflow programming environments that integrate these supportive tools. We therefore focus on performance monitoring, instrumentation and analysis of Grid scientific workflows. In the following section, we describe the two basic concepts behind our supporting Grid workflows: *workflows in the Grid* and *scientific workflows.*

### 3.5.1 Workflows in the Grid

The concept of workflow has been widely used in the business community. The Workflow Management Coalition [13] defines the workflow in the context of business as follows.

**Definition 3.6 (Workflow).** *"Workflow is the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules". [13]*

A Grid workflow can be defined as an automation of a process in the Grid. There are two major differences between Grid workflows and business workflows: (i) a Grid workflow is not limited to a business process, and (ii) a Grid workflow runs on the Grid. These differences rise many challenges to the performance monitoring and analysis. The first difference means that the Grid workflow can be executed in an ad-hoc fashion, not in regular fashion as normally in business workflows. Therefore, monitoring and analysis tools must be capable of handling ad-hoc workflows. The second difference poses many challenges due to the nature of the Grid such as diversity, cross-organization, and dynamics, just to name a few.

- *Diversity*: the monitoring and analysis tool has to collect and utilize a variety of types of data. Moreover, the tool must support diverse environments and must be capable of handling diverse types of applications.
- *Cross-organization*: the monitoring and measurement have to be conducted spanning over multiple organizations. Each organization can impose different policies and may have different requirements.
- *Dynamics*: the resources allocated are frequently changed. Thus, the monitoring and analysis tool has to work on a dynamic manner and has to cope with the dynamics of both Grid applications and resources.

Conventional production and business workflows are normally executed in enterprise environments which are less diverse and dynamic than their Grid counterparts.

### 3.5.2 Scientific Workflows

Most works in the scientific process are collecting, measuring, generating, analyzing, and inferring large amounts of heterogeneous data through various experiments conducted in the process [12, 64, 265]. Therefore, scientific workflows [223, 262, 164, 265] are quite different from business counterparts. Munindar P. Singh and colleagues define scientific workflows as follows.

**Definition 3.7 (Scientific workflows).** *"Scientific workflows describe a series of structured activities and computations that arise in scientific problem-solving". [223, 262]*

Different from production and administrative business workflows, in which the business processes are usually predefined and executed in a routine fashion, scientific workflows are normally more flexible and diverse[3]. Scientific workflows normally connect a variety of scientific analysis tools executed on diverse computational environments. Another aspect is that applications in scientific workflows are typically high-computation demand and performance-oriented. Therefore, performance monitoring and analysis of scientific workflows and applications within the scientific workflows are important tasks.

### 3.5.3 Modeling Workflows and Mapping Workflows onto Grid Environments

A workflow (WF) consists of WF constructs. Each WF construct consists of a set of activities. WF constructs can be fork-join, sequence, do loop, etc. More details of existing WF constructs can be found in [20]. Each activity is associated with one or multiple invoked application(s). Invoked applications can be executed in sequential or parallel manner.

Currently, in our performance monitoring and analysis, we focus on scientific workflows modeled as Direct Acyclic Graph (DAG) workflows.

[3] see [265, 110] for detailed differences between scientific workflows and business workflows

**Definition 3.8 (Workflow model).** *A* workflow *is modeled by a Directed Acyclic Graph (DAG) $WF = (N, E)$, where $N$ is the set of workflow* activities *and $E$ is the set of* directed activity dependencies*. A node $a \in N$ represents an activity (task), a directed edge $(a_i, a_j) \in E$ indicates that the execution of $a_i$ must be finished before the execution of $a_j$.*

The invoked applications, which perform the processing of particular activities, of a workflow are executable programs and shell scripts; shell scripts may call executable programs. The edge represents the control flow only. Data transfer tasks in the workflow are modeled as activities. Hence, we denote $(a_i, a_j)$ as the dependency between activity $a_i$ and $a_j$. Let $G = (N, E)$ be a given DAG and select an arbitrary activity $a_i$. We denote *pred($a_i$)* and *succ($a_i$)* as the sets of the immediate predecessors and successors, respectively, of $a_i$. Formally,

$$succ(a) = \{b \in N | \exists e \in E \text{ such that } e = (a, b)\}$$

$$pred(a) = \{b \in N | \exists e \in E \text{ such that } e = (b, a)\}$$

Figure 3.8 presents the execution sequence of a WF. The user submits a WF to the workflow management system (WfMS) [13]. When executing a workflow, the WfMS instantiates workflow activities. The WfMS locates the Grid sites and submits invoked applications of activity instances to local schedulers of Grid sites. The schedulers locate computational nodes and execute processes of the invoked applications on corresponding nodes.

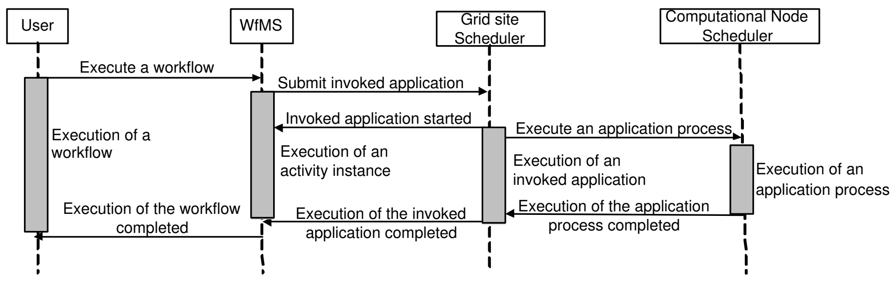

User
WfMS
Grid site Scheduler
Computational Node Scheduler
Execute a workflow
Submit invoked application
Invoked application started
Execute an application process
Execution of a workflow
Execution of an activity instance
Execution of an invoked application
Execution of an application process
Execution of the workflow completed
Execution of the invoked application completed
Execution of the application process completed

**Fig. 3.8.** Execution model of a workflow.

The WfMS manages the execution of the whole workflow. A WfMS may consist of several tools and services such as

- *Process Definition tool*: defines, models and constructs the workflow.
- *Scheduling service*: locates resources for activities and maps activities onto that resources.
- *Workflow Invocation and Control service*: executes activities and controls the execution of activities.
- etc.

We collect monitoring data of activities from the *Workflow Invocation and Control* (WIC) service. This service knows the state of activities as well as the resources on which invoked applications of activities are executed.

### 3.5.4 Workflow Execution Status

WfMS manages the status of activity instances. An *activity state* represents internal conditions determining the status of an activity instance [16]. When an activity instance moves from one activity state to another state, a *state transition* happens. The WfMS records the state transition and generates an *activity event* that captures information related to the state transition.

Our monitoring and analysis tool will monitor and process information about activity states and activity events. Activity states and events are used to build *activity execution status graph.* We use a directed, acyclic, bipartite graph to represent the activity execution status graph of scientific workflows in the Grid.

**Definition 3.9 (Activity execution status graph).** *An activity execution status graph consists of activity states and activity events of an activity instance in a workflow. Let $P$ be an activity execution status graph. $P$ is a directed, acyclic, bipartite graph $(S, E, A)$ in which $S$ is a set of nodes representing activity states, $E$ is a set of nodes representing activity events, and $A$ is a set of edges representing ordered pairs of activity state and event.*

The activity execution status graph, $P(S, E, A)$, may consist of a single activity state or a single activity event. If $a = (x, y) \in A$, then either $(x \in S) \wedge (y \in E)$ or $(x \in E) \wedge (y \in S)$. The activity execution status graph is modeled by using the discrete process presented in [227].

Performance analysis for parallel programs and programs as part of workflow activities are based on the program structure and a classification of code regions which are explained in the following.

## 3.6 Program Structure and Classification of Code Regions

### 3.6.1 Program Structure

Our performance analysis supports programs written in imperative languages such as Fortran, C/C++. In this section we discuss the program structure based on Fortran concepts and terminologies defined in [92].

**Definition 3.10 (Program).** *A program consists of one or more program units.*

A program unit, e.g. a function, procedure (subroutine), is usually a sequence of program statements that define the data environment, the program controls and the computations. Basically, an execution of a program is a sequence of actions. *Program statements* are classified into executable and non-executable statements. An *executable statement* performs or controls one or more actions, e.g., an assignment, a function call. A *non-executable statement* defines the program environment in which actions happen, e.g., a variable declaration. The syntax of a statement in a programming language is defined by the specification of the corresponding language. An *executable construct*, e.g. *do loop, if-then-else, while do, for*, is built up from executable statements.

### 3.6.2 Code Region

The basic elements in our analysis are code regions.

**Definition 3.11 (Code region).** *A code region $r$ is a lexically sequence of program statements of a program unit that is enclosed by two program statements $s_{enter}$ and $s_{exit}$; $s_{enter}$ denotes the first program statement of $r$ and $s_{exit}$ denotes the last program statement of $r$.*

Our instrumentation system can automatically determine typical executable statements and constructs of interest in program units such as loops, function calls, the entire program units, etc., and abstract them as code regions. We call such code regions *tool-defined code regions*. Another type of code regions is called *arbitrary code region* which refers to code regions explicitly defined by the end-user.

**Definition 3.12 (Arbitrary code region).** *An arbitrary code region $r$ is a user-defined code region which consists of program statements enclosed by two arbitrary program statements $s_{enter}$ and $s_{exit}$; $s_{enter}$ denotes the first program statement of $r$ and $s_{exit}$ denotes the last program statement of $r$.*

A program contains a set of instrumented code regions.

**Definition 3.13 (Program code region set).** *A set of (instrumented) code regions in a program version is represented as $CR = \{r_1, r_2, \cdots, r_n\}$.*

### 3.6.3 Classification of Code Regions

Code regions normally have some common properties with respect to the code region type (e.g. a loop or function call) and functionality (e.g. used to send data or to synchronize processes). In order to support the user to select code regions and to support the instrumentation and measurement system to identify potential performance problems occurring in code regions, we classify code regions into subclasses (categories). At the top level, code regions are divided into *generic code regions* and *specific code regions*, as shown in Fig. 3.9.

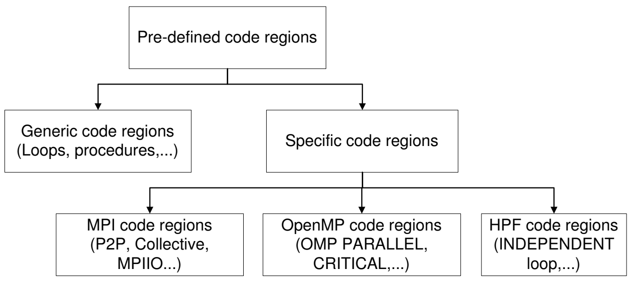

Pre-defined code regions
Generic code regions (Loops, procedures,...)
Specific code regions
MPI code regions (P2P, Collective, MPIIO...)
OpenMP code regions (OMP PARALLEL, CRITICAL,...)
HPF code regions (INDEPENDENT loop,...)

**Fig. 3.9.** The classification of code regions.

Generic code regions are programming paradigm-independent. Subclasses of *generic code regions* are loop, procedure call[4], arbitrary code region, etc. Specific code regions are programming paradigm-dependent code regions which further are classified into:

- *Subclasses of MPI code regions*: send, receive, scatter, broadcast, etc.
- *Subclasses of OpenMP code regions*: parallel, parallel do, critical section, etc.
- *Subclasses of HPF code regions*: executor, independent, etc.

A code region class is assigned to a unique *code region type* which is used to distinguish code region classes. A subclass of code regions might associate with certain types of performance overheads. For example, the subclass of code regions used to send data typically associates with communication overheads. By classifying code regions in a program, given an interesting performance metric, the performance analysis framework can determine code regions that should be instrumented and analyzed in order to obtain that metric.

The current code region classification covers only code regions of invoked applications that we support. Note that although we support workflows, the concept of code regions is not associated with workflows, but associated with invoked applications of workflows. The invoked applications can be sequential, OpenMP, MPI, HPF programs.

A given code region may associate with one or several types of performance problems that need to be determined. In the instrumentation phase, the instrumentation system will detect potential code regions, map them to corresponding code region types, and insert probes to collect performance measurements. In the analysis phase, different code region types will be analyzed differently to compute possible overheads occurred. The list of code region types can be found in Appendix C.1.

[4] Henceforth, we use two terms *procedure* and *subroutine* interchangeably.

### 3.6.4 Processing Unit

A code region can be executed multiple times during runtime of a program. A processing unit indicates the context on which the code region is executed.

**Definition 3.14 (Processing unit).** *A processing unit specifies the context on which a code region is executed. In a parallel application, it is built up from information of* computational node, process, thread *in the experiment. A processing unit pu is a triple* $(n, p, t)$ *where* $n$, $p$ *and* $t$ *are computational node, process identifier and thread identifier, respectively. In a Grid workflow application, it is built up from information of* activity, computational node, process, thread *in the experiment. A processing unit pu is a quadruples* $(a, n, p, t)$ *where* $a$, $n$, $p$ *and* $t$ *are activity identifier, computational node, process identifier and thread identifier, respectively.*

Note that the processing unit does not contain information about Grid sites. We can determine Grid sites based on computational nodes because each computational node is associated with a Grid site.

### 3.6.5 Dynamic Code Region Call Graph (DRG)

Every program consists of a set of *code regions* which can range from a single statement to the entire program unit. A code region can be, respectively, entered and exited by multiple entry and exit control flow points (see Figure 3.10). In most cases, however, code regions are single-entry-single-exit ones.

In order to measure the execution behavior of a code region, the instrumentation system has to detect all *entry* and *exit* nodes of a code region and insert probes at these nodes. Basically, this task can be done with the support of a compiler or guided through manual insertion of directives. Figure 3.10 shows an example of a code region with its entry and exit nodes. To select an arbitrary code region, the user, respectively, marks two statements as the entry and exit statements – which are at the same time entry and exit nodes – of the code region (e.g., by using directives). Through the compiler analysis, the instrumentation system then automatically tries to determine all entry and exit nodes of the code region. Each node represents a statement in the program. Figure 3.10 shows an example of code region with multiple entry and exit nodes. The instrumentation tries to detect all of these nodes and automatically inserts probes before and after all entry and exit nodes, respectively. Code regions can overlap each other, however, we at this point do not support instrumentation of overlapped code regions. Meanwhile we support mainly instrumentation of single-entry multiple-exit code regions. We are investigating to support also multiple-entry multiple-exit code regions.

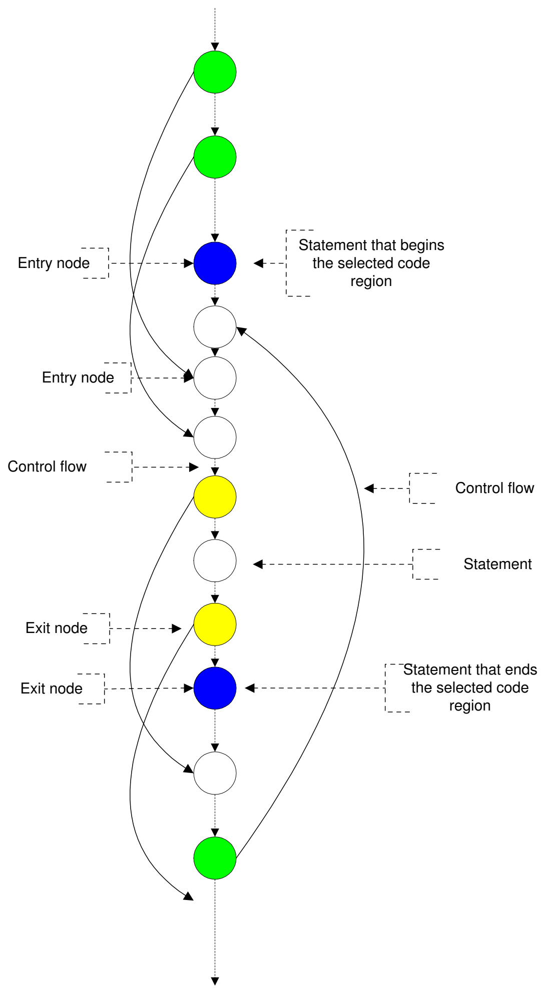

Entry node
Statement that begins the selected code region
Entry node
Control flow
Control flow
Statement
Exit node
Exit node
Statement that ends the selected code region

**Fig. 3.10.** A code region with several entry and exit points

#### 3.6.5.1 Dynamic Code Region Call Graph (DRG)

A code region is *active* (active code region) if its execution has begun but has not yet terminated[5]. An instrumented (static) code region can be executed multiple times (each time one activation of the code region is executed) during runtime of a program. An activation of the code region is also called as a *code region instance*.

**Definition 3.15 (Code region instance).** *An execution of a code region $r$ on a processing unit $pu$ in an experiment $e$ is called a code region instance.*

[5] In other words, the program execution reaches statements of the code region.

One activation of an instrumented code region is associated with a *code region stack trace* which is our extension version of the program stack trace for functions [120] to arbitrary code regions (ranging from a single statement to the entire program unit):

**Definition 3.16 (Code region stack trace).** *A code region stack trace of a program at an instant of time is the sequence of instrumented code regions that are active at that time.*

A code region stack trace $C$ denoted by $C = (c_{r_{m_1}} \rightarrow c_{r_{m_2}} \rightarrow \cdots \rightarrow c_{r_{m_k}})$ starts with an activation $c_{r_{m_1}}$ of the root code region $r_{m_1}$, followed by an activation $c_{r_{m_2}}$ of code region $r_{m_2}$ executed inside the activation $c_{r_{m_1}}$, followed by an activation of code region $r_{m_3}$ executed inside the activation $c_{r_{m_2}}$, and so on. The above definition implies that activations of the same code region may appear in the same code region stack trace or in various code region stack traces. For example, a program $Q$ which has a set of code regions $R = \{r_1, r_2, r_3, r_4\}$ may have three different code region stack traces at different times as follows:

- $c^1_{r_1} \rightarrow c^1_{r_3} \rightarrow c^1_{r_4}$,
- $c^1_{r_1} \rightarrow c^1_{r_2} \rightarrow c^3_{r_3} \rightarrow c^4_{r_4}$, and
- $c^1_{r_1} \rightarrow c^3_{r_2} \rightarrow c^4_{r_3} \rightarrow c^5_{r_4}$

where $r_1$ is the root code region and $c^l_{r_i} \rightarrow c^k_{r_j}$ means that activation numbered $k$ of code region $r_j$ is executed inside activation numbered $l$ of code region $r_i$. We then define the equivalence relation for code region stack traces:

**Definition 3.17 (Equivalence relation for code region stack traces).** *Let $p^{n_k}_{r_{m_k}}$ be the activation numbered $n_k$ of code region $r_{m_k}$ and $q^{n'_{k'}}_{r'_{m_{k'}}}$ be the activation numbered $n'_{k'}$ of code region $r'_{m_{k'}}$. $C_p = (p^{n_1}_{r_{m_1}} \rightarrow p^{n_2}_{r_{m_2}} \rightarrow \cdots \rightarrow p^{n_k}_{r_{m_k}})$ is the code region stack trace which leads to activation $p^{n_k}_{r_{m_k}}$ and $C_q = (q^{n'_1}_{r'_{m_1}} \rightarrow q^{n'_2}_{r'_{m_2}} \rightarrow \cdots \rightarrow q^{n'_{k'}}_{r'_{m_{k'}}})$ is the code region stack trace which leads to activation $q^{n'_{k'}}_{r'_{m_{k'}}}$. The stack trace $C_p$ is called equivalent with $C_q$ iff both stack traces satisfy the following conditions: (i) $k = k'$, $k \geqslant 1$ and (ii) for all $1 \leqslant i \leqslant k$, $r_{m_i} \equiv r'_{m_i}$.*

To clarify the above definition, we consider the above-mentioned example. The equivalence relation holds for the code region stack trace of activation $c^4_{r_4}$ and $c^5_{r_4}$. However, the code region stack trace of activation $c^1_{r_4}$ and $c^4_{r_4}$ do not fulfill the conditions of the equivalence relation. The key idea is that if activations of a code region have equivalent code region stack traces, we can compact them into a record of the data structure representing the calling behavior of code regions while still can associate their performance metrics with their calling context. For example, instead of storing information of activation $c^4_{r_4}$ and $c^5_{r_4}$ in two separate records, we compact their information into one record.

We then extend the code region stack trace into the context of the parallel program with the assumption that a parallel program has multiple processes, each process consists of several threads of computation, and the parallel program is executed on a set of computational nodes.

Each code region stack trace is associated with a *processing unit* on which activations inside the code region stack trace are executed. We can define a new data structure called dynamic code region call graph (DRG) which is used to record the calling behavior and performance metrics of code regions:

**Definition 3.18 (Dynamic code region call graph(DRG)).** *A dynamic code region call graph (DRG) of a program $Q$ with a set of code regions $R = \{r_1, r_2, ..., r_n\}$ is defined by a directed flow graph $G = (N, E, s)$ with a set of nodes $N$ and a set of edges $E$. A node $n \in N$ represents a set of activations of a code region $r_k \in R$ which is executed at least once during runtime of $Q$. The equivalence relation holds for all code region stack traces of all activations in $n$. An edge $(n_1, n_2) \in E$ is a pair of $n_1, n_2 \in N$ where $n_1$ and $n_2$ are a set of activations of code region $r_p$ and $r_q$, respectively and activations in $n_2$ are executed directly inside activations in $n_1$. The set of activations of the first code region executed during execution of $Q$ is defined by $s$.*

In one experiment, a parallel program is executed on multiple processing units. Each DRG of a program is associated with a processing unit. We build a DRG for a performance experiment by creating a *dynamic processing unit graph* (DPG) that includes nodes for computational node, process, and thread. In a DPG, a thread is a child of a process which is a child of a computational node. Each DRG of program is then associated with a thread node of DPG that results in a DRG for a performance experiment. For example, Figure 3.11 shows an excerpt of an OpenMP code together with its associated DRG of performance experiment.

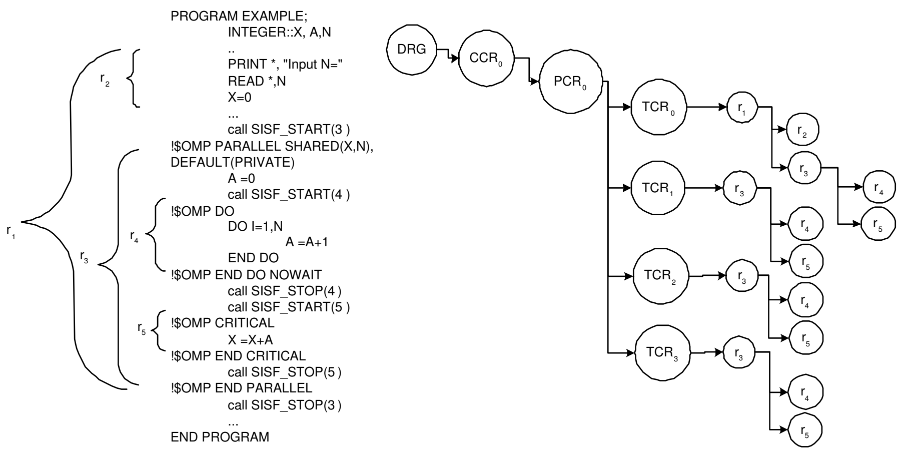
The DRG is stimulated by the idea of the calling context tree (CCT) [26]. However, the DRG differs from CCT in several aspects:

- A node in the CCT represents for activations at procedure-level whereas in the DRG a node is defined as a set of activations of an arbitrary code region (e.g. function, loop, statement, etc.).
- The DRG provides the context of parallel program. Calling context associates with not only call path of code regions but also computational calling environments (computational nodes, processes, threads).

Each activation of a code region is associated with a set of measurements. When constructing a node $n$ of the DRG, measurements of activations in $n$ will be summarized by aggregated operators (e.g. sum). A node in DRG is also called a *dynamic code region summary.*[6]

**Definition 3.19 (Dynamic code region summary).** *A region summary rs is used to store performance metric records of region instances of a code*

[6] Hence *region summary* refers to *dynamic code region summary.*



```
PROGRAM EXAMPLE;
          INTEGER::X, A,N
          ..
          PRINT *, "Input N="
          READ *,N
          X=0
          ...
          call SISF_START(3 )
!$OMP PARALLEL SHARED(X,N),
DEFAULT(PRIVATE)
          A =0
          call SISF_START(4 )
!$OMP DO
          DO I=1,N
                    A =A+1
          END DO
!$OMP END DO NOWAIT
          call SISF_STOP(4 )
          call SISF_START(5 )
!$OMP CRITICAL
          X =X+A
!$OMP END CRITICAL
          call SISF_STOP(5 )
!$OMP END PARALLEL
          call SISF_STOP(3 )
          ...
END PROGRAM
```

**Fig. 3.11.** OpenMP code excerpt with DRG when executing with 4 threads (denoted by $TCR_0 - TCR_3$). Code regions $r_1, r_2$ are executed only in thread 0 whereas $r_3, r_4, r_5$ are executed in 4 threads. This program is running with 1 process (denoted by $PCR_0$) on a computational node (denoted by $CCR_0$).

*region r in a processing unit pu; the instances must have the same code region stack trace.*

The DRG is used as a key data structure to conduct a detailed performance overhead analysis under SCALEA (presented in Chapter 5). Note that the generic timing overhead of a code region $r$ with $n$ explicitly instrumented sub-regions $r_1, ..., r_n$ is given by

$$T(r) = T(Start_r) + T(r_1) + ... + T(r_n) + T(Remain) + T(End_r) \qquad (3.1)$$

where $T(r_i)$ is the timing overhead for an explicitly instrumented code region $r_i$ ( $1 \leq i \leq n$). $T(Start_r)$ and $T(End_r)$ correspond to the overhead at the beginning (e.g. fork threads, redistribute data, etc.) and at the end (join threads, barrier synchronization, process reduction operation, etc.) of $r$. $T(Remain)$ corresponds to the code regions that have not been explicitly instrumented. However, we can easily compute $T(Remain)$ as region $r$ is instrumented as well.

Call graph techniques have been widely used in performance analysis. Tools such as Vampir [179], gprof [112, 89], CXperf [126] support a call graph which shows how much time is spent in each function and its children. In [55] a call graph is used to improve the search strategy for automated performance diagnosis. The DRG requires less space than the dynamic call tree (each node represents a single activation of a procedure) and provides information more precisely than the dynamic call graph (each node represents all activations of a procedure). In addition, nodes of the call graph in these tools represent function calls [89, 126]. In contrast our DRG defines a node as an arbitrary code region (e.g. function, function call, loop, statement).

#### 3.6.5.2 Generating and Building the Dynamic Code Region Call Graph

Calling code region $r_2$ inside a code region $r_1$ during the execution of a program establishes a parent-children relationship between executions of $r_1$ and $r_2$. The measurement library will capture these relationships, build nodes of the DRG and maintain them during the execution of the program. Each node in the DRG is represented by a data entry point which contains performance measurement for all activations whose code region stack traces are equivalent. An edge $(n_1, n_2)$ of the DRG is represented by a link from the caller $n_1$ to the callee node $n_2$. In our implementation, a global pointer per thread is used to maintain the current node (data entry point) of the sub-DRG of the thread. The performance measurement of the current activation will be stored into the corresponding data entry point maintained by the global pointer.

If a new activation of code region $r_2$ starts to be executed inside an activation of code region $r_1$, the instrumentation library then searches all children of the current data entry point maintained by the global pointer. If no child representing activations of $r_2$ is found, then a new data entry point is generated for recording information of the new activation and an edge linking the current entry point to the new data entry point is made. Otherwise, there exists a previous data entry point representing activations of $r_2$; code region stack traces of these activations are equivalent to that of the new activation. In this case, the existing data entry point is used to keep information of the new activation. In both cases, the global pointer is then pointed to the new or existing data entry point which holds information of the new activation. When the new activation ends its execution, the data entry point maintained by the global pointer will be updated with performance measurement. And then, the global pointer is transfered to the parent of the current data entry point. If a code region $r$ (e.g. the first code region to be executed) is encountered that it isn't child of any other code region, an abstract code region is assigned as its parent. Every code region has a unique identifier which is included in the probe inserted by the instrumentation system and stored in the *instrumentation description file* (presented in Section 5.3.4).

The DRG data structure maintains the information of code regions that are instrumented and executed. Every thread of each process will build and maintain its own sub-DRG when executing. In the post-processing phase, the DRG of the entire application will be built based on the individual sub-DRGs of all threads by processing the profiles/trace files that contain the performance data of threads.

## 3.7 Performance Experiments

To conduct the performance analysis of an application, we execute the application on various resources and collect performance measurements for analysis task. The results of the analysis are used as inputs to various tasks, e.g. performance tuning, parameter study, performance prediction, etc.

**Definition 3.20 (Performance experiment).** *Each run of an application on a set of given resources is called an experiment.*

Each performance experiment is a procedure in which input variables of an execution of an application are purposefully changed so that we may observe and identify the corresponding changes in the performance of the application by studying the performance measurements of the execution. Figure 3.12 presents the simple black-box model (also known as experiment process model) [177, 8] which can be used to describe experiments we will encounter in performance measurement and analysis. The application processes will be stimulated by factors. These factors can either be controlled (such as problem size or machine settings) or uncontrolled (such as operating system load). These factors interact with application processes. Measuring execution of application processes produces performance measurements. These performance measurements can be used to analyze the application processes that we measure. Through the analysis phase, performance metrics that reflect the performance of the application are produced.

**Definition 3.21 (Performance metric).** *A performance metric reflects the quantity of some specific performance behavior.*

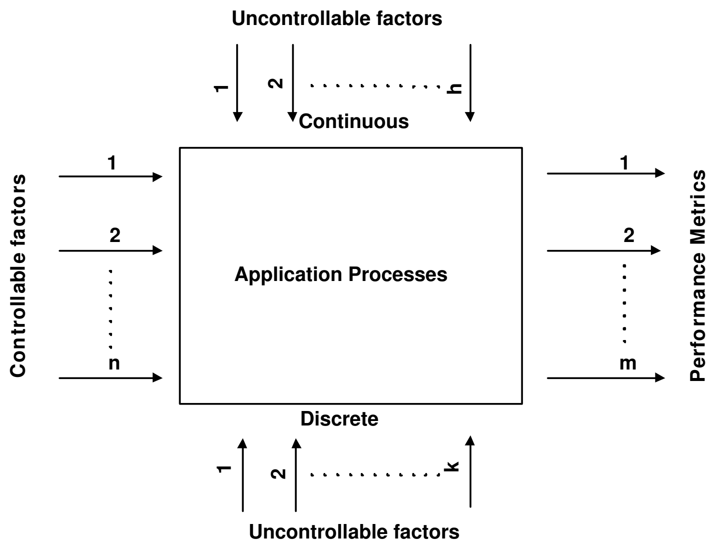


**Fig. 3.12.** General model of a performance experiment.

Factors and performance metrics obtained are used to explain the performance of the application. The factors and performance metrics are measured, collected and analyzed in order to observe and understand how they behave and relate to each other. Performance data collected in each experiment is organized into a *performance experiment data*.

For each performance study, a set of *performance metrics* (performance criteria) is selected. Not all metrics in that set can be obtained in the measurement phase. Instead, some metrics will be obtained in measurement phase, others will be available only after or during the analysis phase. The choice of performance metrics is an important issue and is a difficult task [137]. We categorize the performance metrics into:

- *measured timing metrics*: timing metrics obtained from the measurement. Examples of measured timing metrics are wall-clock time, system time, etc.
- *measured counter metrics*: counter metrics obtained from the measurement. Measured counters also include hardware counters. Examples of measurement counters metrics are the number of L2 cache accesses, the number of function calls, etc.
- *overhead metrics*: metrics derived from overhead analysis (presented in Chapter 5). Examples of overhead metrics are send/receive, synchronization overheads, etc.
- *ratio metrics*: metrics which are computed from other metrics by using ratio-based scheme. Examples of ratio metrics are L2 cache miss ratio, system time per wall-clock time, etc.

Values of performance metrics are quantitative data. As pointed in [137], depending on the utility function of a performance metric, the metric can be categorized into three classes:

- *higher is better*: the higher value of a metric indicates the better performance with respect to the metric. Examples of this type of metrics in our tool are cache hit ratio, MFLOPS (Millions of Floating-Point Operations Per Second).
- *lower is better*: the lower value of a metric indicates the better performance. Examples of this type of metrics in our tool are overhead metrics, response time, etc..
- *nominal is best*: both high and low values are undesirable, depending on the analysis and the client of the analysis. Examples of this type of metrics are resource utilization, the average number of jobs in the queue, etc.

Through the measurement and analysis, a set of performance metrics will be measured and analyzed for every code regions. Performance metrics of a code region are stored in *performance metric records*.

**Definition 3.22 (Performance metric record).** *A performance metric record $pm$ is represented as a tuple $(m, v)$ where $m$ is the metric name and $v$ is the metric value.*

**Definition 3.23 (Performance class).** *Given a performance metric, the performance metric can be classified into classes. Each class is associated with a function that maps the value of the performance metric into the class.*

For example, we can classify L2 cache miss ratio into three classes: *poor, average, good*. The class *good* can be determined as follows

$$f_{good}(x) = \begin{cases} 1 & x \leq 0.3, \\ 0 & x > 0.3 \end{cases} \tag{3.2}$$

where $x$ is the value of L2 cache miss ratio and $f_{good}$ is the function that maps the value of L2 cache miss ratio into the class *good*. Each performance class is identified by a unique *performance characteristic term*. Performance class can be used to represent the qualitative aspect (e.g., good or poor) of performance metrics.

**Definition 3.24 (Performance characteristic term).** *A performance characteristic term is a linguistic term that determines a performance class.*

Examples of performance characteristic terms are *good, poor, high*, etc.

Although the performance of a code region is characterized by a set of performance metrics associated with the code region, in this dissertation, the term *performance of a code region* refers to wall-clock time of the code region. Similarly, the *performance of a parallel or Grid application* refers to the wall-clock time of the application. When mentioning the performance of a code region with respect to a specific metric, we indicate the metric name. Note that the performance of the entire program can be determined through the performance of a code region (the entire main unit of the program). The execution time of the whole main unit of a program is the execution time of the whole program.

**Definition 3.25 (Performance experiment data).** *Performance data collected in an experiment is called* performance experiment data*. Performance experiment data $E$ can be described as $E = (PU, RS, CR)$ where $PU$ is a processing unit set, $RS$ is region summary set, and $CR$ is (instrumented) code region set.*

## 3.8 Summary

Our methods and techniques of performance monitoring, instrumentation, measurement and analysis are targeting to OpenMP, MPI, HPF and hybrid parallel programs on SMP cluster and Grid environments and scientific workflows on the Grid.

Code regions are classified in order to support the user to select code regions and to support the instrumentation and measurement system to detect the performance problem occurring in a code region. We introduce a new representation of code regions, called the dynamic code region call graph (DRG). The DRG reflects the dynamic relationship between code regions and its subregions and stores performance measurements of code regions in a compact form. The DRG is not restricted to function calls but also covers loops, I/O and communication statements, etc. We have presented the performance experiment model which is used to study relationships between experiment factors and performance metrics and to analyze the performance of applications.

# 4 Introduction to SCALEA-G

## 4.1 Introduction

In this chapter we first introduce SCALEA which is a performance analysis framework for parallel programs that covers HPF, OpenMP, MPI and mixed programming paradigms. We then describe the SCALEA-G framework which unifies performance monitoring and analysis of Grid infrastructures and applications, and dynamic instrumentation of applications in the Grid. SCALEA-G consists of distributed monitoring and performance sensors and sensor manager services, directory services for supporting information publication and discovery, archival system for storing collected monitoring and performance data, instrumentation service, and various performance analyzers. SCALEA-G is built up on a set of open Grid protocols and partially reuses existing state-of-the art tools and framework for Grid, and the SCALEA system. SCALEA-G is a middleware that is capable of self-management. It exploits the concept of Grid Monitoring Architecture (GMA) [30], sensor networks [23, 257] and peer-to-peer computing [173], and is being implemented as a set of OGSA-enabled grid services [93, 94].

## 4.2 SCALEA Overview

SCALEA is a performance instrumentation, measurement, and analysis system for distributed memory, shared memory, and mixed parallel programs. The main objective of SCALEA is to support performance oriented program development and to help the programmer to understand the intricate details of the programming model, program transformation system, and architecture that may result in any performance problem. Figure 4.1 shows the architecture of SCALEA which consists of several components: SCALEA Instrumentation System (SIS), SCALEA Runtime System, SCALEA Performance Data Repository, and SCALEA Performance Analysis & Visualization System.

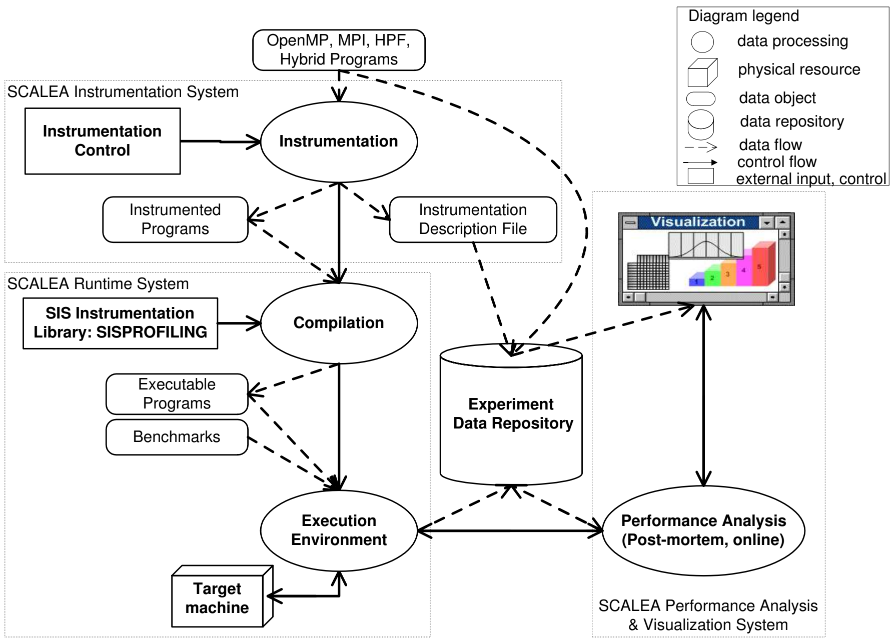


**Fig. 4.1.** Architecture of SCALEA

SIS is an automatic instrumentation system for Fortran programs. SIS is built based on the Vienna Fortran Compiler (VFC) [37] which is a compiler that translates HPF/HPF+ Fortran programs into Fortran90/MPI or mixed OpenMP/MPI programs. The parser and unparser of VFC can work with MPI, OpenMP, OpenMP/MPI, HPF/HPF+ Fortran 90 programs. SIS utilizes the parser and unparser of VFC for processing the input programs. Thus, it supports HPF/HPF+, MPI, OpenMP and hybrid parallel programs, e.g. OpenMP/MPI and HPF+/OpenMP. The input programs of SIS are processed by the compiler frontend which generates an abstract syntax tree (AST). SIS enables the user to select (by directives or command-line options) code regions and performance metrics (timing, hardware counters, and performance overheads) of interest. Moreover, SIS provides an interface for other tools to traverse and annotate the AST to specify code regions for which performance metrics should be obtained. Based on pre-selected code regions and/or performance metrics, SIS automatically inserts probes (instrumentation code) in the code. The probes will collect all relevant performance information during the execution of the program on the target architecture. SIS also generates an *instrumentation description file* (presented in Section 5.3.4) that enables to relate all gathered performance data to the input program. We also work on a prototype of SIS based on Open64 open sources [5].

The SCALEA runtime system supports profiling [242] for cluster applications and uses benchmarks to capture performance data of individual computational nodes and networks in clusters. Profiling measurement is provided by SISPROFILING library which collects timing and counter information, including hardware counters, of code regions. Hardware counters are determined through an interface with the PAPI library [49].

After compiling and linking the instrumented program with the SCALEA instrumentation library, an executable program is created. Depending on the instrumentation library linked, profile files storing performance data will be created during the execution of the program. The post-mortem performance analysis comprises two phases. First, a pre-processing phase filters and extracts all relevant performance information from the profile files and computes the DRG. Both filtered performance data and DRG are then stored in an experiment data repository. Second, a performance overhead analysis is conducted that reads performance data from the data repository, analyzes them, computes the performance overhead requested, and stores them in the repository. The performance overhead analysis can also analyze performance data in profile files. Moreover, a visualization engine is provided that displays various metrics in isolation or together with the source code.

The experiment data repository is used to store experiment-related data, e.g., performance data and result, application source files, etc. The repository is powered by PostgreSQL[10] which is supported on many platforms.

SCALEA provides analysis tools interacting with the user via a rich set of Graphic User Interface (GUI) components. SCALEA Analysis and Visualization components are written in Java and access the PostgreSQL database during the analysis. Therefore, the analysis can be carried out on any platform that supports Java. This makes SCALEA to be more portability than other tools that are running only on a specific platform.

## 4.3 SISPROFILING

SISPROFILING contains a set of measurement libraries which are used to capture performance measurements of cluster and grid applications.

On the cluster environment, SISPROFILING supports post-mortem analysis by providing measurement libraries for measuring sequential, OpenMP, MPI, HPF, and hybrid parallel programs written in Fortran and C/C++. These libraries capture performance data and store that data into files, which later on are processed by performance analysis tools, for instance SCALEA.

On the Grid environment, SISPROFILING supports online analysis by providing application sensors that measure and gather performance information of Grid applications and middleware. These sensors capture monitoring data and performance measurements and send the data to the online analysis components.

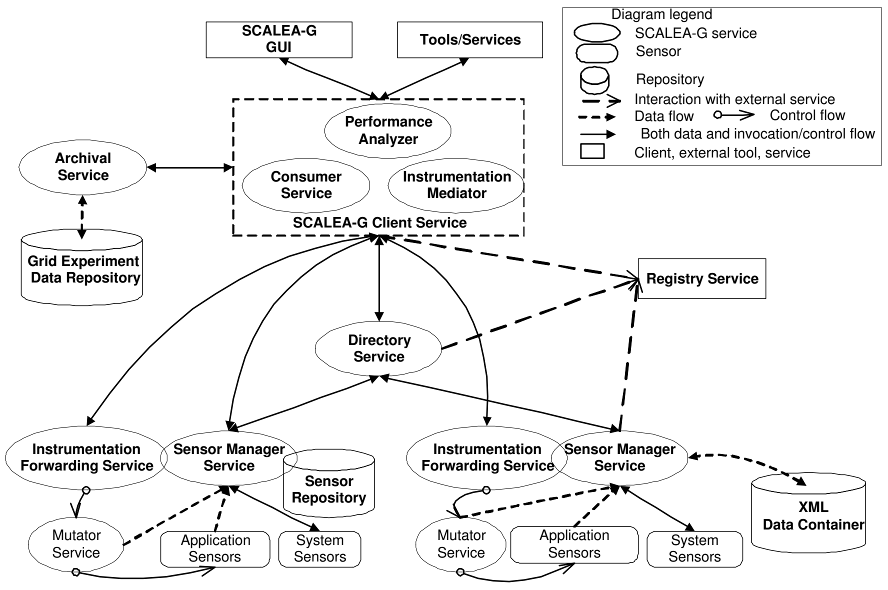


**Fig. 4.2.** High-level view of SCALEA-G architecture

## 4.4 SCALEA-G

SCALEA-G is a unified monitoring and performance analysis system for the Grid. SCALEA-G is an open architecture that consists of a set of Grid services and sensors for monitoring and performance analysis of Grid infrastructures and applications. Figure 4.2 depicts the architecture of SCALEA-G framework which includes the following main components: Directory Service (DS), Sensor Manager Service (SM), Mutator Service (MS), Client Service (CS), and Graphical User Interface (GUI). All of them are Grid-enabled services/clients and can be deployed on different hosting environments. DS is the core service that provides information about data in SCALEA-G. SM is the core service for archiving and providing performance and monitoring data, and MS is the core service for dynamic instrumentation of Grid applications.

The *Directory Service* (DS) is used for publishing and searching information about producers and consumers that produce and consume performance data, and information about types and characteristics of that data.

The *Archival Service* (AS) is a data repository which is used to store performance results collected and analyzed by other components.

The *Sensor Manager Service* (SM) is used to manage sensors that gather and/or measure a variety of types of data for monitoring and performance analysis, to store monitoring data and to provide that data to consumers. Application instrumentation can be done at source code level manually or automatically, or dynamically at the runtime; source code instrumentation is based on SIS. The *Instrumentation Forwarding Service* (IFS) receives instrumentation requests from clients and forwards the requests to the *Mutator Service* (MS) which conducts the dynamic instrumentation.

The *Client Service* (CS) provides interfaces for administrating other SCALEA-G services and for accessing data in these services. In addition, it provides functions for analyzing performance data. Any external tools and services can access SCALEA-G by using CS.

A *GUI* - supported by CS - enables the user to graphically examine monitoring and performance analysis results. SCALEA-G services register and search information about their service instances in *Registry Services.*

Interactions among SCALEA-G services are divided into *Grid service-based operations* and *TCP-based stream data delivery.* Grid service operations are used to perform tasks which include controlling activities of services and sensors, subscribing and querying performance data, registering, querying and receiving information from DS. TCP-based stream channels are used to transfer monitoring data, performance data and results among producers (e.g. sensors, SMs) and consumers (e.g. SMs, clients). Grid service operations incorporate transport-level and message-level security whereas TCP channels are based on secure connections; the security in SCALEA-G relies on Grid Security Infrastructure (GSI) [267].

When deploying SCALEA-G, instances of sensors and MSs are executed on computational nodes being monitored. An instance of SM can be deployed to manage multiple sensors and MSs in a node or a set of nodes, depending on the real system. Similarly, SMs in different administrative domains can publish information in multiple DS instances. The client discovers SCALEA-G services through Registry Services which can be deployed in different domains.

The main objective of our approach is to unify a variety of types of monitoring and performance data. While we have a set of different sensors to provide different data types, whose data structures are diverse, we have to make sure that the clients use the same interface with the same mechanism to access that diverse data types.

## 4.5 The Role of SCALEA/SCALEA-G in ASKALON Toolset

The design and implementation of SCALEA/SCALEA-G is largely centered on the tool integration aspect. SCALEA/SCALEA-G are targeted to two types of clients: the end-user and the external tools/services. SCALEA/SCALEA-G are parts of the ASKALON programming environment and toolset for cluster and Grid architectures [88]. ASKALON integrates four interoperable tools: SCALEA/SCALEA-G for instrumentation, monitoring and performance analysis, ZENTURIO for automatic experiment management [201], AKSUM for automatic bottleneck analysis [87], and the PerformanceProphet for application modeling and performance prediction [195]. Figure 4.3 presents the

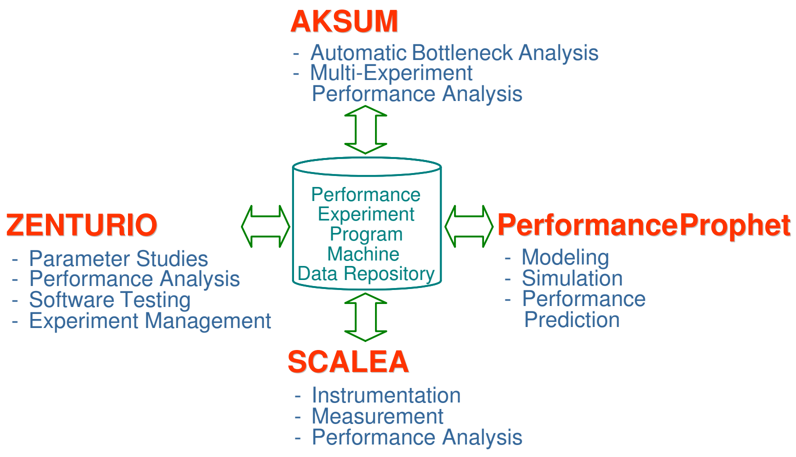


**Fig. 4.3.** ASKALON tool integration.

architecture of ASKALON in which SCALEA, ZENTURIO, ASKUM and PerformanceProphet are integrated into a single framework.

SCALEA is used by various other tools in ASKALON to support automatic bottleneck analysis, performance experiment and parameter studies, and performance prediction of parallel programs. In this context, SCALEA provides functions for instrumentation, measurement and analysis of parallel programs, collects relevant experimental data including application, source code, machine information, and performance data and results, and stores collected data and information into the repository. SCALEA provides an interface with search and filter capabilities for other tools to access the repository. Other tools such as such as AKSUM, PerformanceProphet and ZENTURIO can access data in the repository through the provided interface. Data can also be exported into XML format so that it can easily be transfered to and processed by other tools. SCALEA-G is currently being integrated with new Grid scheduler, resource brokering, and workflow management service being developed recently in the ASKALON toolset.

## 4.6 Summary

SCALEA, which supports performance instrumentation, measurement , analysis and visualization of parallel programs, and SCALEA-G, a unified system for performance monitoring, data integration and analysis for the Grid, are the two main frameworks in which proposed techniques and methods in this dissertation have been developed and applied. SCALEA and SCALEA-G are parts of ASKALON toolset for cluster and grid computing that support various other high-level tools such as experiment managements, performance predictions, automatic bottleneck search.

# 5

# Performance Analysis for Parallel Programs

## 5.1 Introduction

In this chapter we present a novel classification of temporal overhead for parallel programs that is used to compute a variety of performance metrics. A highly flexible instrumentation and measurement system is provided which can be controlled by command-line options and program directives. We present a library which can be interfaced by external tools through the provision of a full Fortran90 OpenMP/MPI/HPF frontend. The library allows external tools to instrument an abstract syntax tree at a very high-level with C-function calls and to generate source code.

We exploit a relational-based experiment data repository for performance analysis. We describe the design and use of SCALEA's experiment data repository which is employed to store information about performance experiments including application, source code, machine information and performance information. The performance results are associated with experiments, source code and machine information. SCALEA is able to offer search and filter capabilities, supporting performance analysis and comparison of multiple experiments. Moreover, the repository provides well-defined interfaces for other tools to access the experiment data, thus leveraging the performance data sharing and tool integration.

We present a new approach to automatic performance analysis that we call the *soft performance analysis*. In this approach, we use soft computing techniques, such as fuzzy logic (FL), machine learning (ML) concept, and the combination of FL and ML, and similarity theory to develop soft techniques and methods for performance analysis of parallel and distributed programs.

This chapter is based on the work presented in [255, 244, 254, 245, 246, 247].

## 5.2 Classification of Temporal Overheads

According to Amdahl's law [25], theoretically the best sequential algorithm takes time $T_s$ to finish the program, and $T_p$ is the time required to execute the parallel version with $p$ processors. The temporal overhead of a parallel program, $T_o$, is defined by

**Definition 5.2.1 (Temporal overhead)** $T_o = T_p - T_s/p$

The temporal overhead reflects the difference between achieved and optimal parallelization. $T_o$ can be divided into $T_i$ and $T_u$ such that $T_o = T_i + T_u$, where $T_i$ is the overhead that can be identified and $T_u$ is the overhead fraction which could not be analyzed in detail. Theoretically, $T_o$ can never be negative, which implies that the speedup $T_s/T_p$ can never exceed $p$ [156]. However, in practice it occurs that temporal overhead can become negative due to super linear speedup of applications. This effect is commonly caused by an increased available cache size.

Temporal overhead can be classified into subclasses, each describes one type of overhead. Figure 5.1 shows the latest version of our novel and substantially refined overhead classification which includes *Data movement, Synchronization, Control of parallelism, Additional computation, Loss of parallelism* and *Unidentified overhead*.

### 5.2.1 Data Movement

Data movement shown in Fig. 5.1(b) corresponds to any data transfer within local memory (e.g. cache misses and page faults), file I/O, communication (e.g. point-to-point or collective communication), and remote memory access (e.g. put and get).

The communication overhead is classified into point-to-point and collective communication overhead. Point-to-point communication overhead is due to the network overhead of exchanging data between two peers such as send and receive operations. Collective communication overhead includes collective operations such as scatter, broadcast operations and the communication part of collective operations.

Remote memory access (RMA) overhead is due to operations on memory in remote machines including *put* and *get* (e.g. MPI RMA communication). Note that the *Communication of Accumulate Operation* overhead has been stimulated by the `MPI_Accumulate` construct [115] which is employed to move and combine (through MPI reduction operations) data at a remote site via remote memory access.

Table 5.1 presents an example of code regions which may cause data movement overhead, and measurement methods.

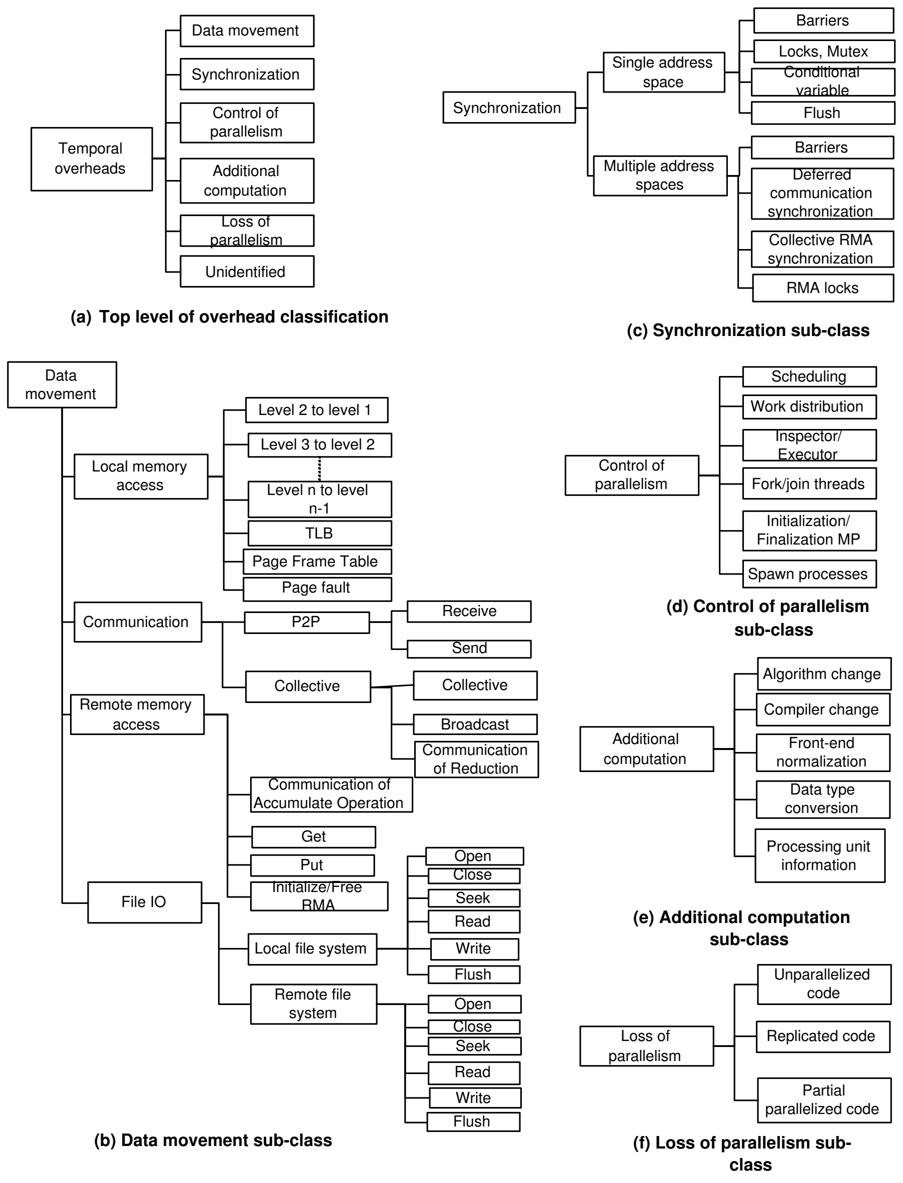


**Fig. 5.1.** The classification of temporal overhead.

| **Subclass of Data Movement Overhead** | **Code Regions** | **Measurement Methods** |
|---|---|---|
| Local memory access | All | Hardware counter measurement, benchmark, etc. |
| Communication | `MPI_SEND, MPI_RECV, MPI_ISEND, MPI_SENDRECV, MPI_BCAST, MPI_GATHER,MPI_SCATTER, MPI_REDUCE`, etc. | Timer, Benchmark |
| Remote memory access | `MPI_PUT, MPI_GET,MPI_ACCUMULATE`, etc. | Timer |
| File IO | `MPI_FILE_READ, MPI_FILE_WRITE`, etc. | Timer |

**Table 5.1.** Overhead of data movement: example of code regions and measurement methods.

### 5.2.2 Synchronization

*Synchronization* (e.g. barriers and locks) shown in Fig. 5.1(c) is used to coordinate processes and threads when accessing data, maintaining consistent computations and data, etc. We subdivided the synchronization overhead into single- and multiple-address space overheads. A single-address space overhead corresponds a synchronization inside a single process on parallel systems, for instance any kind of OpenMP synchronization falls into this category whereas multi-address space synchronization has been stimulated by MPI synchronization, RMA locks, barriers between different processes, etc.

Table 5.2 presents an example of code regions that may cause synchronization overhead, and methods that are used to measure the synchronization overhead.

### 5.2.3 Control of Parallelism

*Control of parallelism* (e.g. fork/join operations and loop scheduling) shown in Fig. 5.1(d) is used to control and manage the parallelism of a program that can be due to code inserted by the compiler (e.g. runtime library) or by the programmer (e.g. to implement data redistribution).

Table 5.3 presents an example of code regions that may cause control of parallelism overhead, and methods that are used to measure the overhead.

### 5.2.4 Additional Computation

*Additional computation* shown in Fig. 5.1(e) reflects any change of the original sequential program including algorithmic or compiler changes to increase par-

| **Subclass of Synchronization Overhead** | **Code Regions** | **Measurement Methods** |
|---|---|---|
| Single Address Space Synchronization | `OMP BARRIER, OMP FLUSH, OMP_INIT_LOCK, OMP_SET_LOCK`, etc. | Timer, Benchmarks |
| Multiple Address Space Synchronization | `MPI_BARRIER, MPI_WIN_FENCE, MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST, MPI_WIN_WAIT, MPI_WIN_UNLOCK, MPI_WIN_UNLOCK,MPI_WAIT, MPI_TEST`, etc. | Timer, Benchmark |

**Table 5.2.** Synchronization overhead: example of code regions and measurement methods.

| **Subclass of Control of Parallelism Overhead** | **Code Regions** | **Measurement Methods** |
|---|---|---|
| Schedule | `OMP SCHEDULE`, etc. | Timer, Benchmark |
| Work distribution | HPF INDEPENDENT | Timer |
| Inspector/Executor | HPF INDEPENDENT | Timer |
| Fork/join threads | `OMP PARALLEL, OMP PARALLEL DO`, etc. | Timer, Benchmark |
| Initialization/Finalization message passing | `MPI_INIT, MPI_FINALIZE`, etc. | Timer |
| Spawn processes | `MPI_COMM_SPAWN_MULTIPLE, MPI_COMM_SPAWN`, etc. | Timer, Benchmark |

**Table 5.3.** Control of parallelism overhead: example of code regions and the measurement methods.

allelism (e.g. by eliminating data dependences) or data locality (e.g. through changing data access patterns). Moreover, requesting the processing unit identification or the number of threads to execute a code region can also imply additional computation overhead.

Table 5.4 presents an example of code regions that may cause additional computation overhead, and methods that are used to measure the overhead.

### 5.2.5 Loss of Parallelism

*Loss of parallelism* shown in Fig. 5.1(f) is due to imperfect parallelization of a program. That is a workload in sequential version that has not been conducted fully in parallel in the parallel version.

**Definition 5.1 (Workload).** *The workload of an application is the total amount of work that the application must conduct.*

| **Subclass of Additional Computation Overhead** | **Code Regions** | **Measurement Methods** |
|---|---|---|
| Algorithm change | all | Timer |
| Compiler change | all | Timer |
| Front-end normalization | all | Timer |
| Data type conversion | `MPI_PACK, MPI_UNPACK` | Timer |
| Processing unit information | `MPI_CART_CREATE`, `OMP_IN_PARALLEL`, etc. | Timer |

**Table 5.4.** Additional computation overhead: example of code regions and measurement methods.

Loss of parallelism is further classified into unparallelized code (executed by only one processor), replicated code (executed by all processors), and partially parallelized code (executed by more than one but not all processors).

Table 5.5 presents an example of code regions that may cause loss of parallelism overhead, and methods that are used to measure the overhead.

| **Overhead Category** | **Code regions** | **Measurement Method** |
|---|---|---|
| Unparallelized code | ALL | Timer |
| Replicated code | ALL | Timer |
| Partial parallelized code | ALL | Timer |

**Table 5.5.** Loss of parallel overhead: example of code regions and measurement methods.

### 5.2.5.1 Unparallelized code

Let $r$ be a code region. Let $T_p(r)$ be the execution time of $r$ in the experiment with $p$ processors. Let $W_r$ be the total amount of work of $r$. If this code is executed in parallel with $p$ processors, theoretically $W_r/p$ will be the amount of work executed on each processor in the ideal case, thus each processor should take $T_p(r)/p$ to complete $r$. However, if $r$ is an unparallelized code region, only one processor executes it on behalf of all others. The loss of parallelism $T_{olopa}$ due to unparallelized code $r$ executed in the processor executes $r$ is defined by

$$T_{olopa}(r) = \frac{(p-1)T_p(r)}{p} \tag{5.1}$$

If $T_p(r)$ is fixed, it means that the execution time of the unparallelized code does not depend on the number of executing processors, the overhead of unparallelized code will increase when the number of processors is increased.

**Remark 1** *Loss of parallelism overhead due to sequential code region will increase when the number of processors is increased.*

*Proof:* We compare the unparallelised overhead when using $p$ and $p'$ processors. Let $T_{olopa}(r)$ and $T'_{olopa}(r)$ be the overhead due to unparallelized code in $p$ and $p'$ processors, respectively. Let $p \leq p'$, we have

$$T_{olopa}(r) = \frac{(p-1)T_p(r)}{p} \tag{5.2}$$

$$T'_{olopa}(r) = \frac{(p'-1)T_{p'}(r)}{p'} = \frac{(p'-1)T_p(r)}{p'}, \tag{5.3}$$

$$\begin{aligned} T_{olopa}(r) \leq T'_{olopa}(r) &\equiv \frac{(p-1)T_p(r)}{p} \leq \frac{(p'-1)T_p(r)}{p'} \\ &\equiv (p \times p' - p') \leq (p' \times p - p) \equiv p \leq p' \square \end{aligned} \tag{5.4}$$

*Note 5.2.* Theoretically, if $r$ is an unparallelized code region then $T_p(r) = T_s(r)$ where $T_s(r)$ is the execution time of $r$ in sequential version. Thus, Equation 5.1 can be rewritten as

$$T_{olopa}(r) = \frac{(p-1)T_s(r)}{p} \tag{5.5}$$

However, we opt Equation 5.1 because there are unparallelized code regions which exist in parallel version but not in sequential version. Therefore, Equation 5.1 must be more generic.

#### 5.2.5.2 Replicated code

It is the case when the same work is carried out in parallel in the parallel version. Theoretically, let $W_r$ be the workload conducted by a code region $r$. In principle, with $p$ processors, each processor should take $W_r/p$. However, if $r$ is a replicated code region, processors which execute $r$ will take $W_r$. Therefore, a loss of parallelism overhead due to replicated code region $r$ in processor $i$, $T^i_{olopa}(r)$, can be defined by

$$T^i_{olopa} = T^i_p(r) - \frac{T^i_p(r)}{p} \tag{5.6}$$

where $T^i_p(r)$ are the execution time of $r$ in processor $i$ of the parallel version with $p$ processors.

#### 5.2.5.3 Partial parallelized code

It is the case when the work is carried out in parallel but not enough for all processors. An overhead due to partial parallelized region $r$ can be measured as follows:

$$T^i_{olopa}(r) = T^i_p(r) - \frac{\sum_{i=1}^{w} T^i_p(r)}{p}, 1 < w < p \tag{5.7}$$

where $T_p^i(r)$ is the execution time of code region $r$ on processor $i$, $T_{olopa}^i(r)$ is the loss of parallelism due to partial parallelized code, $p$ is the total processors allocated for conducting the computation of $r$ and $w$ is the actual number of processors executing this code region.

### 5.2.6 Unidentified Overhead

*Unidentified overhead* corresponds to the overhead that is not covered by the above categories.

### 5.2.7 A Note on the Classification of Temporal Overhead

In theory it would be desirable to provide an orthogonal overhead classification as suggested in [68]. In practice, however, it is very difficult to build an overhead analysis that can support an orthogonal overhead classification. For instance, data movement commonly contains synchronization overhead. Control of parallelism may contain synchronization and data movement, etc. It is however of paramount importance that any overhead is assigned to only one sub-class of the overhead classification scheme. For example, if we can measure synchronization time as part of a data movement operation, then we should count this synchronization time either for data movement or for synchronization overhead but not for both of them.

## 5.3 The Instrumentation System

SCALEA Instrumentation System (SIS) provides the user with three alternatives to control instrumentation which includes command-line options, directives, and an instrumentation library combined with a Fortran90 OpenMP/ HPF/MPI frontend and unparser. All of these alternatives allow the specification of performance metrics and code regions of interest for which SCALEA automatically generates instrumentation code and determines the desired performance values during or after program execution. In the remainder of this section we assume that a code region refers to a single-entry single-exit code region. A large set of predefined mnemonics (for a detailed list see Appendix C.1) is provided by SIS for selecting code regions and performance metrics. The current implementation of SIS supports a variety of code region and performance metric mnemonics:

- code region mnemonics: arbitrary code regions, loops, outermost loops, procedures, I/O statements, HPF INDEPENDENT loops, HPF redistribution, OpenMP parallel loops, OpenMP Sections, OpenMP Critical, MPI send, receive, and barrier statements, etc.

- performance metric mnemonics: wall-clock time, CPU time, communication overhead, cache misses, barrier time, synchronization, scheduling, compiler overhead, unparallelized code overhead, hardware parameters, etc. See also Fig. 5.1 for a classification of performance overheads considered by SIS.

### 5.3.1 Command-line Options

With the command-line options, performance metrics and code regions of interest for instrumentation are specified through the command-line parameters when invoking the instrumentation system. Command-line options can be used along with directives. SIS provides a large set of command-line options for instrumenting parallel programs (see Appendix D.1 for more details about command-line options).

### 5.3.2 SIS Directives

The user can specify arbitrary code regions ranging from the entire program units to single statements, as shown in the following:

**!SIS$ CR** *region_name* **BEGIN**
*code region r*
**!SIS$ END CR**

Command-line options and other SIS directives (mentioned below) can then be used to indicate whether or not arbitrary code regions will be instrumented and if so, what performance metrics should be computed for them.

In order to specify a set of code regions $R = \{r_1, ..., r_n\}$ in an enclosing region $r$ and performance metrics which should be computed for every region in $R$, SIS offers the following directive:

**!SIS$ CR** *region_name* [,*cr_mnem-list*] [**PMETRIC** *perf_mnem-list*] **BEGIN**
*code region r that includes all regions in R*
**!SIS$ END CR**

Code region $r$ defines the *scope* of the directive. Note that every (code) region in $R$ is a sub-region of $r$ but $r$ may contain sub-regions that are not in $R$ as well.

The code region (*cr_mnem-list*) and performance metric (*perf_mnem-list*) mnemonics are indicated as a list of mnemonics separated by commas. One of the code region mnemonics (CR_A) refers to arbitrary code regions. Note that the above specified directive allows indicating either only code region mnemonics or performance metric mnemonics, or a combination of both. If in a SIS directive $d$ only code region mnemonics are indicated, then SIS is instrumenting all code regions that correspond to these mnemonics inside of the scope of *d*. The instrumentation is done for a set of default performance metrics which can be overwritten by command-line options. This option can be particularly useful if the user knows that only a few code regions can cause critical performance problems whose performance overheads are unknown. Only if the mnemonic CR_A is included in the list of code region mnemonics of a directive *d*, then instrumentation for arbitrary code regions inside of the scope of *d* will be conducted.

If only performance metric mnemonics are indicated in a directive *d* then SIS is instrumenting those code regions that have an impact on the specified metrics. This option is very useful if a user is interested in specific performance metrics but does not know which code regions may cause these overheads. For instance, compilers often substantially restructure OpenMP (e.g. implicit synchronization) and HPF (e.g. implicit data redistribution) code which is not visible at the input code. In order to find the associated compiler overhead, the programmer, e.g., could specify some control of parallelism overhead mnemonics in a SIS directive.

If both code region and performance metrics are defined in a directive *d*, then SIS is instrumenting these code regions for the indicated performance metrics in the scope of *d*. Feasibility checks are conducted by SIS, for instance, to determine whether the programmer is asking for OpenMP overheads in HPF code regions. Warnings are displayed to the programmer. If neither code region nor overhead mnemonics are indicated then the directive is simply ignored.

All previous directives are called local directives as the scope of these directives is restricted to part of a program unit (main program, subroutines or functions). The scope of a directive can be extended to the entire program unit by using the following syntax:

**!SIS$** [**CR** *cr_mnem-list*] [**PMETRIC** *perf_mnem-list*]

A global directive *d* collects performance metrics – indicated in the `PMETRIC` part of *d* – for all code regions – specified in the `CR` part of *d* – in the program unit which contains *d*. A local directive implies the request for performance information restricted to the scope of *d*. There can be nested directives with arbitrary combinations of global and local directives. If different performance metrics are requested for a specific code region by several nested directives, then the union of these metrics is determined.

SIS also supports the user to map a code region to a code region type.

**!SIS$ CR** *coderegiontype*, *coderegionname* **BEGIN**
*code region r*
**!SIS$ END CR**

This feature allows the user to specify a set of different code regions at different locations as a single code region. Thus the instrumentation and measurement will aggregate performance measurements of different code regions under a single record. Furthermore, SIS provides specific directives in order to control tracing/profiling. The directives MEASURE ENABLE and MEASURE DISABLE allow the programmer to turn on and off tracing/profiling of specific code regions.

> **!SIS$ MEASURE DISABLE**
> *code region r*
> **!SIS$ MEASURE ENABLE**

In the following example we demonstrate some of the directives as mentioned above by showing a fraction of the Stommel application [189].

```
d1: !SIS$ CR PMETRIC ODATA_SEND, ODATA_RECV, ODATA_COL
    call MPI_BCAST(nx, 1,MPI_INTEGER, mpi_master,MPI_COMM_WORLD,mpi_err)
    ...
d2: !SIS$ CR comp_main, CR_A, CR_S PMETRIC wtime, L2_TCM BEGIN
        ...
d3:     !SIS$ CR init_comp BEGIN
             dj=real(nx,b8)/real(nodes_row,b8)
             ...
d4:     !SIS$ END CR
             ...
d5:     !SIS$ MEASURE DISABLE
             call bc(psi,i1,i2,j1,j2)
d6:     !SIS$ MEASURE ENABLE
             ...
             call do_force(i1,i2,j1,j2)
             ...
d7: !SIS$ END CR
```

Directive $d_1$ is a global directive which instructs SIS to instrument all send, receive and collective communication statements in this program unit. Directives $d_2$ (begin) and $d_7$ (end) define a specific code region with the name *comp_m ain*. Within code region *comp_main*, SCALEA will determine wall-clock times (*wtime*) and the total number of L2 cache misses (*L2_TCM*) for all arbitrary code regions (based on mnemonic *CR_A*) and subroutine calls (mnemonic *CR_S*) as specified in $d_2$. Directives $d_3$ and $d_4$ specify an arbitrary code region with the name *init_comp*. No instrumentation as well as measurement is done for the code region between directives $d_5$ and $d_6$.

### 5.3.3 SIS High-level Instrumentation Library

SIS has been integrated with the VFC [37] source-to-source compiler which supports a Fortran90 OpenMP/MPI/HPF frontend and provides an abstract syntax tree and a sophisticated mechanism to traverse the AST, to detect code patterns and to generate source code. Even though directives and command line options are powerful mechanisms for requesting instrumentation, high-level tools must select code regions and insert directives and/or invoke command lines. In order to faciliate the interaction between the high-level tool and the instrumentation system, a high-level instrumentation library is provided.

SIS provides a high-level library that can be used by other tools to exploit SIS instrumentation functions. We have developed a C-library that can be used to traverse the AST and to mark arbitrary code regions for instrumentation; the library is called SISHL. For each code region, the high-level tool can specify the performance metrics of interest. Similarly to SIS directives, SISHL provides corresponding functions to insert directives into source code. The detailed APIs of SISHL can be found in Appendix D.2.

For example, in order to instrument all *loops* (denoted by `CR_L`) and *subroutine calls* (denoted by `CR_S`) enclosed by statement $s1$ and $s2$ and to indicate the instrumentation to measure metrics wall-clock time (denoted by `wtime`) and L2 cache misses (denoted by `L2_TCM`), the high-level tool can use the following code:

```
char *perf_mnem[5]={"wtime","L2_TCM"};
char *cr_mnem[5]={"CR_L","CR_S"};
sishl_insert_local(s1,s2, cr_mnem,2,perf_mnem,2,"region1'');
```

SISHL will traverse the AST and insert instrumentation directives used to measure the requested code regions and metrics. Based on the annotated AST, SIS then automatically generates an instrumented source code.

### 5.3.4 Instrumentation Description File

A crucial aspect of performance analysis is to relate performance information back to the original input program. When instrumenting a program, SIS generates an *instrumentation description file* (IDF) which correlates profiling, trace and overhead information with the corresponding code regions. For every instrumented code region, the IDF maintains a variety of information (see Table 5.6).

A code region type describes the type of the code region, for instance, entire program, outermost loop, read statement, OpenMP SECTION, OpenMP parallel loop, MPI barrier, etc. The program unit corresponds to a subroutine or function which encloses the code region. The performance data of a code region stored in a separate repository is associated with the code region via the code region identifier.

### 5.3.5 Standard Interface for an Instrumentation Engine

In Section 5.3.3, we present how the other tools can use high-level instrumentation APIs of SIS to request the instrumentation system to instrument and measure performance metrics and code regions of interest. SIS takes the source of programs, represents programs in its own intermediate language (IL), and

| IDF Entry | Description |
|---|---|
| id | code region identifier |
| type | code region type |
| file | source file identifier |
| unit | identifier of the program unit which encloses this region |
| line_start | line number where this region starts |
| column_start | column number where this region starts |
| line_end | line number where this region ends |
| column_end | column number where this region ends |
| pm_mnemonics | mnemonics representing performance metrics which will be collected or computed for this region |
| aux | auxiliary information |

**Table 5.6.** Contents of an entry in the instrumentation description file (IDF)

constructs the AST. SIS provides APIs for traversing and querying performance metrics and code regions based on the IL (e.g. VFC IL or WHIRL [15]) it uses. The high-level tool (e.g. AKSUM [87]) then uses these APIs to traverse and query the request.

However, this approach has some limitations. The high-level tool has to know the IL used by the instrumentation system and to use the APIs which are dependent on the IL. As a result, the high-level tool is likely to stick to the infrastructure provided by a single instrumentation. As the high-level tool usually does not need detailed information about the source code and should not be fixed to a specific underlying instrumentation system, the instrumentation system can provide a simplied and standardized mechanism for the high-level tool to specify the instrumentation requests. To this end, we can:

- standardize the way to describe instrumentation requests (e.g. using directives and command options).
- provide a high-level and simple IL (e.g. based on XML).
- standardize instrumentation description file (for relating performance measurements back to source code).

The high-level tool uses the high-level IL and commands to control and request the instrumentation system. The instrumentation engine agrees on the way to specify instrumentation requests (e.g. by directive or command lines), and provides the high-level IL. The high-level IR should be simple as much as possible. However, it should provide enough capabilities for traversing and manipulating source code at level of *program units, executable constructs, executable statements*; it may not consider checking semantics of source programs.

This idea has been realized by the joint work conducted in APART working group [1]. A high-level IL, based on XML, and a set of instrumentation and monitoring requests have been proposed to facilitate the interaction between the high level tool and the instrumentation system [219, 218].

### 5.3.6 Measurement Probes

In order to collect and provide measurements for the performance analysis of a parallel program, firstly code regions in that program are selected, and then instrumented and measured. The selection and instrumentation can be done automatically by tools, as presented in previous sections. To measure a code region, appropriate *probes* are inserted into the source code. A generic probe in SIS has the following form

```
sis_start(PB, cr_type)
      code region r
sis_stop(PB)
```

where $PB$ points to a unique internal data structure used to store measurement data of the code region $r$; $PB$ also associates with a unique data entry in the instrumentation description file; the entry contains information about the instrumented code region. *cr_type* is code region type according to the classification of code regions. Probes will be started at entry points and stopped at exit points of code regions. Performance metrics measured can be controlled at the instrumentation phase by directives or at runtime by setting environment variables. In the latter case, the user can choose timing in combination with hardware counters metrics for each experiment without recompiling the program.

## 5.4 Overhead Analysis based on the DRG

Types of overheads and portion of identified overhead within total overhead determined are dependent on the instrumentation and measurement. In the analysis phase, we conduct two tasks: (1) to determine the total overhead $T_o$, and (2) to determine detailed types of overheads for each code region. After that we can derive the identified overhead $T_i$ and the unidentified part $T_u$.

Given a code region $r$, let $T_s(r)$ and $T_p(r)$ be the execution time in sequential version and in parallel version with $p$ processors, respectively. The total overhead $T_o(r)$ of code region $r$ when executed with $p$ processors can be computed as follows

$$T_o(r) = T_p(r) - \frac{T_s(r)}{p} \tag{5.8}$$

The total overhead can be determined only when (i) both sequential and parallel version of $r$ exist[1], or (ii) $r$ is an addition parallel programming-dependent code region. In the latter case, $r$ is necessarily required for the parallelization of programs, e.g. a code region used to send data (e.g. `MPI_SEND`); $r$ is not introduced in the sequential version.

[1] However, with the complexity of parallel programs, we may not have the sequential version of a corresponding parallel one.

The total overhead provides an overall figure of how much overhead occurs in a code region; with total overhead we may figure out whether a performance problem exists or not. However, in order to determine more detailed information about the sources that contribute on the overall figure, we need to determine sub overhead categories of the total overhead. These categories can provide us more insightful information, which can be used to reveal the causes of performance problems. To this end, we may need to divide a code region into subregions, to measure that subregions and to determine types of overheads of individual subregions besides measuring the code region. To clarify this idea, consider the example presented in Figure 3.11. Based on the DRG, we can compute $T_o(r_3)$, $T_o(r_4)$ and $T_o(r_5)$. Assume that we have the control of parallelism overhead $T_{octrp}(r_4)$ of OMP DO, then we can compute the control of parallelism overhead $T_{octrp}(r_3)$ of OMP PARALLEL as follows:

$$T_{octrp}(r_3) = T_{octrp}(r_4) + T_{octrp}(CR\_OMPBPA) + T_{octrp}(CR\_OMPEPA) \tag{5.9}$$

where $T_{ctrp}(CR\_OMPBPA)$ and $T_{ctrp}(CR\_OMPEPA)$ are the control of parallelism overhead caused by OMP PARALLEL and OMP END PARALLEL directive, respectively. To determine detailed factors of the total overhead, we base on performance measurements, types of code regions. In some cases, training sets data may be employed.

The overhead is analyzed for every thread (it also means for every processors). Figure 5.2 describes the algorithm used to compute overhead based on the DRG. Overhead is determined for all code regions. This algorithm basically computes overhead of threads of a performance experiment. As an experiment is executed on multiple processing units, we may need to define the overhead for the entire experiment. We first compute the overhead for a process based on the overhead of its threads. Let $T_o^i$ and $T_p^i$ be the overhead and execution time of the thread $i$ of a process $P$ with $p$ threads, respectively. Let thread 0 be the master thread. $T_p^0$ is the execution time the process $P$. As threads of a process are executed in parallel, the overhead of a process is not a sum of the overhead of threads, but is a statistics of that. Overhead of process $P$ is computed based on statistics operations, e.g. min, max and average of $\{T_o^i\}$. Similarly, overhead of an experiment is computed by applying statistics operations, e.g. min, max and average, of overhead $T_o$ of processes.

## 5.5 Performance Data Repository for Performance Analysis

Collecting and archiving performance data are vital tasks for performance analysis and optimization process, especially for supporting multi-experiment analysis, performance comparison and automated performance diagnosis. However, little effort has been done to employ data repositories to organize and store performance data for performance analysis. This lack of a systematic organization of data has hindered several aspects of performance analysis

```
startNode ← root node of DRG of program
Call overhead_computing(startNode) to compute overheads of startNode.
procedure overhead_computing(startNode)
begin
    if (startNode is a leaf node) then
      Determine types of overhead categories of startNode.
      Compute total overhead $T_o$ based on the code region type and the equivalent code region in sequential version by applying the equation $T_o = T_p - \frac{T_s}{p}$.
      Compute identified overhead $T_i$ and unidentified overhead $T_u$ of startNode.
    else
      for (all edges e(startNode,currentNode) in DRG) do
        Call overhead_computing(currentNode) to compute overheads of currentNode.
        Compute and update types of overhead categories of startNode based on overheads of currentNode and on type of the code region represented by currentNode.
      end for
      Compute total overhead $T_o$ of startNode based on its code region type and on its equivalent code region in sequential version by applying the equation $T_o = T_p - \frac{T_s}{p}$.
      Compute identified overhead $T_i$ and unidentified overhead $T_u$ of startNode.
    end if
end
```

**Fig. 5.2.** Algorithm for computing overheads.

tools. For example, the users commonly create their own performance data collections, extract performance data, and use external tools to compare performance outcome of several experiments manually. Moreover, different performance tools employ different performance data formats and these tools lack well-defined interfaces for accessing the data they provide. As a result, the collaboration among performance tools and high-level tools is hampered.

Utilizing a data repository and providing well-defined interfaces to access data in the repository can help to overcome the above-mentioned limitations. We can structure the data associated with performance experiments. Therefore, performance results can always be associated with their source codes and description of machine on which the experiment has been taken. Based on that, any other performance tool can store performance data it collects for a given application to the same repository, thus providing a large potential to enable more sophisticated performance analyses. And then, other tools or system software can easily access the performance data through a well-defined interface. To this end, we have investigated and exploited a relational-based experiment data repository for performance analysis.

In this section we describe the approach to exploit a relational-based experiment data repository for performance analysis. We present the design and use of the experiment data repository which is used to store performance data and information about performance experiment which alleviates the association of performance information with experiments and source code. Besides the utilization of powerful search capability of relational database for searching performance data, other two significant achievements have been gained when utilizing this data repository. Firstly, the experiment data repository supports multi-experiment analysis that allows the user to examine and compare the performance outcome of different program executions. Historical performance comparison can be conducted easily. Secondly, the experiment data repository simplifies the collaboration among performance tools and higher-level tools by providing open interfaces for accessing data in the repository, and leverages the performance data sharing and exchanging by exporting performance data into XML format.

### 5.5.1 Performance Metrics Catalog

Most performance and monitoring tools provide various types of performance metrics. However, the lack of documentation and interpretation has prevented the user from understanding the semantics of these performance metrics, and hampered the collaboration among the performance tools and external tools. In order to provide a user/tool a solid understanding of the semantics of performance metrics provided by a given tool, these performance metrics should be documented and standardized.

Performance metrics vary from low level metrics, the metrics provided at the measurement (e.g. wall-clock time, CPU time , hardware counter [49]), to the high-level ones, the metrics derived from low-level metrics (e.g. data movement overhead [246], load imbalance, later send [82]). To document them, we propose each performance metric should be described by, at least, following properties:

- *unique name*: identifies the performance metric.
- *data type*: describes data type (e.g. *double, uint64*) of the value of the performance metric.
- *unit*: indicates the measurement unit (e.g. *microsecond, kilobyte, MB/s*) of the performance metric.
- *well-defined meaning*: explains the semantics of the performance metric. It has to be well defined so that any users, tools can understand the semantics and the usage of the performance metrics they are dealing with.

All defined metrics then can be stored into a *performance metrics catalog*. Based on this catalog, a tool $A$ has necessary information, e.g. name, units, data type of metrics when it wants to utilize performance metrics provided by tool $B$. The performance metrics catalog also helps a tool to understand the semantics of metrics provided by another tool and to make use of metric values correctly.

We have documented such a performance metrics catalog in SCALEA. Table 5.7 shows a sample extracted from SCALEA's performance metrics catalog[2]. This catalog has provided a convenience to the user/tool when specifying, acquiring and retrieving performance data during the instrumentation and analysis procedure. The full list of performance metrics covered by SCALEA can be found in Appendix C.2.

| Name | Data type | Unit | Descriptions |
|---|---|---|---|
| wtime | double | microsecond | Wall-clock time |
| utime | double | microsecond | User CPU time |
| stime | double | microsecond | System CPU time |
| L2_TCM | uint64 | counter | Total level 2 cache misses |
| L2_TCA | uint64 | counter | Total Level 2 cache accesses |
| odata_send | double | microsecond | Overhead due to SEND operations |
| odata_recv | double | microsecond | Overhead due to RECV operations |
| odata_p2p | double | microsecond | Overhead due to send+receive operations |
| odata_col | double | microsecond | Overhead of collective communication |
| octrp_fkjn | double | microsecond | Overhead of Fork/join threads |
| octrp_infl | double | microsecond | Overhead of Initialize/Finalize message passing |

**Table 5.7.** Sample of SCALEA's performance metrics catalog.

### 5.5.2 Experiment Data Repository

The experiment data repository is used to store the most important information about performance experiments including application, source code, machine information, and performance results.

#### 5.5.2.1 Experiment-related data

Figure 5.3 shows the structure of experiment-related model for the data stored in the experiment data repository. An *experiment* refers to a sequential or parallel execution of a program on a given target architecture. Every experiment is described by *experiment-related data*, which includes information about the application code, the part of a machine on which the code has been executed, and performance information. An application (program) may have a number of implementations (code versions), each of them consists of a set of source files and is associated with one or several experiments.

Every source file has one or several static code regions (ranging from entire program units to single statements), uniquely specified by their positions –

[2] In this dissertation, we use the two metric names *system CPU time* and *system time* interchangeably. The two metric names *system CPU time* and *system time* are also used interchangeably.

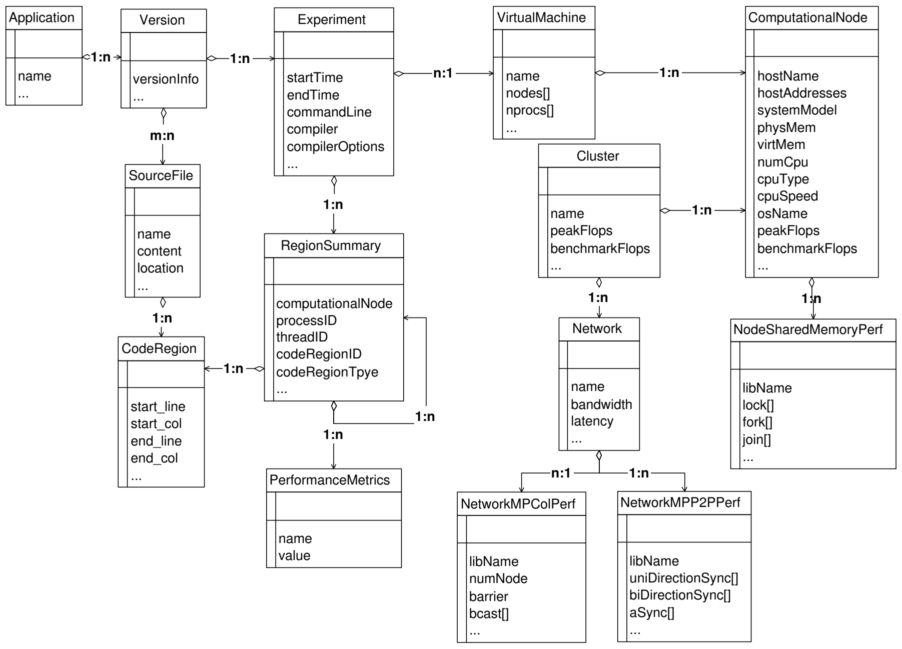


**Fig. 5.3.** SCALEA experiment-related data model

start/end line and column – where the region begins and ends in the source file. A static (instrumented) code region contains following information:

- *region identifier*: a unique static identifier generated or assigned by instrumentation system. The region identifier is used to related performance data to the source code.
- *source positions*: positions (e.g. line, column) of the code region in the source file.
- *code region type*: the type indicates the subclass of the classification of code regions that the code region belongs to (see Section 3.6.3). Depending on the code region type, other static information can be stored, for example, SCHEDULE strategy in OpenMP do loop.

At the runtime, (static) instrumented code regions will be executed. A *region summary* refers to the performance information collected for a given code region on a specific processing unit (see Section 3.6.4). The region summaries are associated with performance metrics that comprise performance overheads, timing information, and counters (including hardware counters). A region summary has a parent region summary; this reflects the dynamic code region relationship. Each *performance metric* is represented in a *tuple (name,value)* where *name* is defined in the performance metrics catalog.

Experiments are associated with the virtual machines on which they have been taken. The virtual machine is a collection of physical machines available to the experiment; it is described as a set of computational nodes (e.g. single-processor systems, SMP nodes) which are gathered in clusters. A *Cluster* is a group of physical machines (computational nodes) that are connected by specific networks. Physical machines in the same cluster have the same physical configuration. Note that our data structure is still suitable for network of workstations which can be classified into small groups of machines having the same configuration. Specific data of physical machines such as memory capacity, peak MFLOPS, etc., are measured and stored the data repository. In addition, for each computational node, performance characteristics of shared memory (e.g. lock, barrier, fork/join thread using shared memory libraries) are measured by benchmarks and stored in *NodeSharedMemoryPerf*. Similarly, performance characteristics for message passing model of networks of a cluster are also measured and stored in *NetworkMPColPef* and *NetworkMPP2PPerf* for collective and point-to-point operations, respectively.

#### 5.5.2.2 Collect, Store and Access Data in Experiment Data Repository

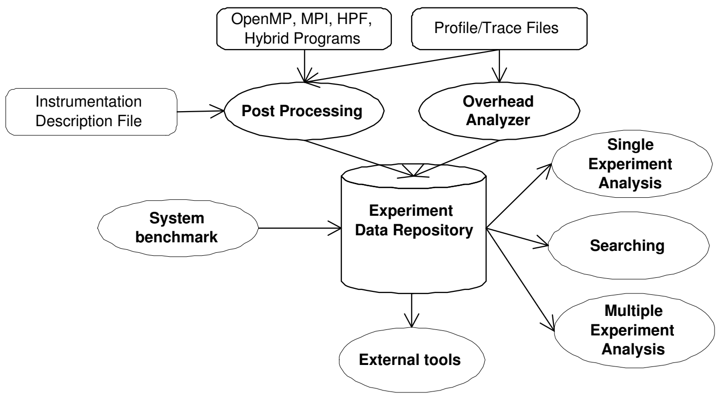

OpenMP, MPI, HPF, Hybrid Programs
Profile/Trace Files
Instrumentation Description File
Post Processing
Overhead Analyzer
Single Experiment Analysis
System benchmark
Experiment Data Repository
Searching
Multiple Experiment Analysis
External tools

**Fig. 5.4.** Components interacting with the experiment data repository.

Figure 5.4 depicts components that interact with the experiment data repository. The *post-processing* is used to store source programs and instrumentation description file generated by instrumentation system into the data repository. In addition, it filters raw performance data collected for each experiment and stores filtered data into the repository.

The *overhead analyzer* performs the overhead analysis according to the overhead classification based on the filtered data in the repository (or based on profile files). The resulting overhead is then stored into the data repository.

The *system benchmark* is used to collect system information (computational node, memory, harddisk, etc), to determine specific information (e.g.

overhead of probes, time to access a lock), and to perform benchmarks for every target machine of interest. By using MPI, OpenMP micro benchmarks, we obtain reference values for both message passing (e.g. blocking send/receive) and shared memory operations (e.g. lock, barrier) with various networks, libraries. The collected data is used for correlation analysis between application- and system-specific metrics. Usually, this task is performed only when a new hardware/software, library version to be installed.

Based on the data available in the repository, various analyses can be conducted such as single- and multi-experiment analysis, performance search, etc.

### 5.5.2.3 Experiment-related data APIs

For each table represented in Fig. 5.3, a corresponding Java class which implements methods to access data in the table is generated. In addition, we define two classes *ProcessingUnit* and *ExperimentData*. The first class is used to describe where the code region is executed; it consists of information of *computational node, process, thread*. The latter implements major interfaces used to access experiment data. Figure 5.5 highlights some classes with few selected methods.

The experiment-related data APIs simplify the way to access performance data in the experiment repository. For example, we can compute the ratio of identifier overhead to wall-clock time, $\frac{oall\_ident}{wtime}$, of code region `region1` in node `gsr1.vcpc.univie.ac.at`, process `1`, thread `0` by querying identified overhead (denoted by *oall_ident*) and wall-clock time (denoted by *wtime*) as follows:

```
CodeRegion cr = new CodeRegion(‘‘region1’’);
Experiment e = new Experiment(‘‘experiment1’’);
ProcessingUnit pu = new ProcessingUnit(‘‘gsr1.vcpc.ac.at’’,1,0);
ExperimentData ed = new ExperimentData(new DatabaseConnection(...));
RegionSummary  rs = ed.getRegionSummary (cr,pu,e);
PerformanceMetric overhead=rs.getMetricOfSummary(’’oall_ident’’);
PerformanceMetric wtime =rs.getMetricOfSummary(‘‘wtime’’);
double overheadRatio=((Double)overhead.metricValue).doubleValue()/
        ((Double)wtime.metricValue).doubleValue();
```

We found that providing standard APIs is one of the keys bringing to the success of the collaboration among tools. Even though different performance tools use different internal data representations, with well-defined APIs for acquiring performance data, the collaboration among tools will be more simple and efficient.

### 5.5.2.4 Exporting Performance Data to XML

Most data in the repository can be exported into XML format so that it can be easily accessed by other tools. For example, performance data of a code

```
public class PerformanceMetric {
  public String metricName;
  public Object metricValue;
  ...
}
public class ProcessingUnit {
  ...
  public ProcessingUnit(String node,int process, int thread) {...}
  ...
}
public class RegionSummary {
  ...
  //get values of metrics measured/computed for regionsummary rs
  public PerformanceMetric getMetricsOfSummary(){...}
  //get value of metric metricName in region summary rs
  public PerformanceMetric getMetric(String metricName){...}
  ...
}
public class ExperimentData {
  DatabaseConnection connection;
  ...
  //get list of processing units of Experiment e.
  public ProcessingUnit[] getProcessingUnit(Experiment e){...}
  //get list of summaries of code region cr in experiment e.
  public RegionSummary[] getRegionSummaries(CodeRegion cr, Experiment e){
  ...}
  //get list of region summaries of code region cr in experiment e and
  //associates with proces sing unit pu.
  public RegionSummary[] getRegionSummaries(CodeRegion cr, Experiment e,
        ProcessingUnit pu, RegionSummary parent) {...}
  //get list of region summaries called inside the input region summary
  public RegionSummary[] getChildOfRegionSummary(RegionSummary rs){...}
  //get the region summary which calls the input region summary rs.
  public RegionSummary getParentOfRegionSummary(RegionSummary rs){...}
  ...
}
```

**Fig. 5.5.** Generated classes for accessing experiment data.

region can be expressed in XML format, as shown in Figure 5.6. The tool which is processing the XML-based performance data can easily parse and use the data by using an XML parser and information in the performance metrics catalog.

```
<coderegion>
     ...
     <metrics>
        <metric name=''wtime'' value=''1.09039995E8'' />
        <metric name=''odata_send'' value=''2986000.0'' />
        <metric name=''odata_recv'' value=''5.6923546E7'' />
        <metric name=''octrp_infl'' value=''4708181.0'' />
     </metrics>
</coderegion>
```

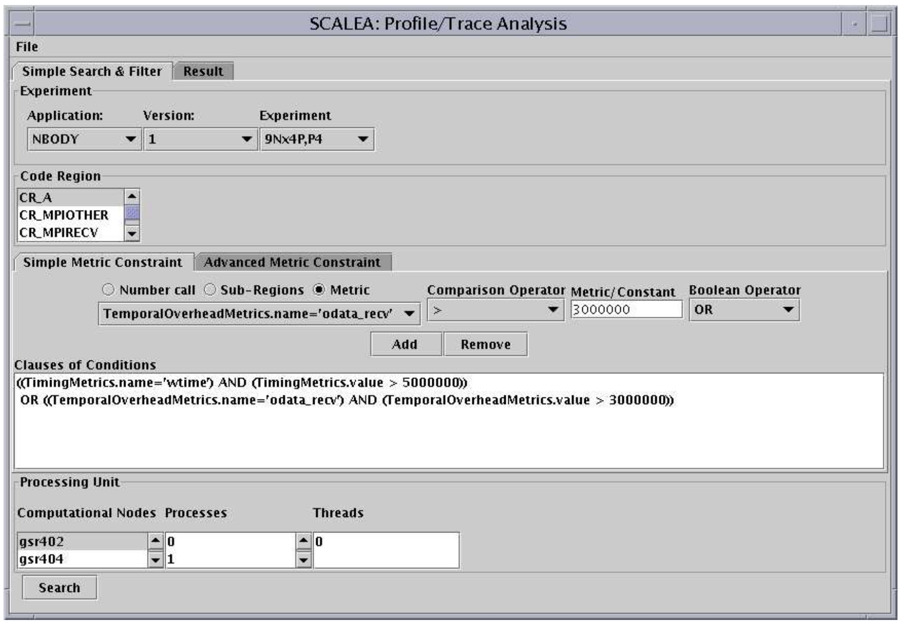
**Fig. 5.6.** Example of extracted performance data in XML.

### 5.5.3 Performance Analysis and Tool Integration based on Experiment Data Repository

In this section, we present three main achievements gained from the use of the experiment data repository named: search and filter capabilities, multi-experiment analysis, and data sharing and tool integration.

#### 5.5.3.1 Search and Filter Capabilities

Most existing performance tools lack basic search and filter capabilities. Commonly, existing tools allow the user to browse code regions and their related performance metrics through various views of performance data. For example, tools such as [279, 179, 258] provide numerous displays including process time-lines with zooming and scrolling, histograms of state durations and message data. Those displays are crucial but they require all data to be loaded into the memory so that the system turns to be in-scalable. Moreover, too much information provided so that the user is difficulty to find out the occurrence of events with criteria performance metrics, e.g. code regions with wall-clock time larger than 20% of the wall-clock time of the program. Search of performance data will allow the user to quickly identify interesting code regions. Filtering performance data being visualized will increase the scalability of performance tools. However, search and filter data on files are not efficient, robust and fast. They require high implementation cost. To employ a relational-based data repository allows us to archive, search and filter data with a great flexibility and robustness based on database management utilities and SQL language, and to minimize the implementation cost.



Experiment

| Application: | Version: | Experiment |
|---|---|---|
| NBODY | 1 | 9Nx4P,P4 |

Code Region: CR_A, CR_MPIOTHER, CR_MPIRECV

Simple Metric Constraint | Advanced Metric Constraint

Number call, Sub-Regions, [✓ Metric]: TemporalOverheadMetrics.name='odata_recv'

| Comparison Operator | Metric/Constant | Boolean Operator |
|---|---|---|
| > | 3000000 | OR |

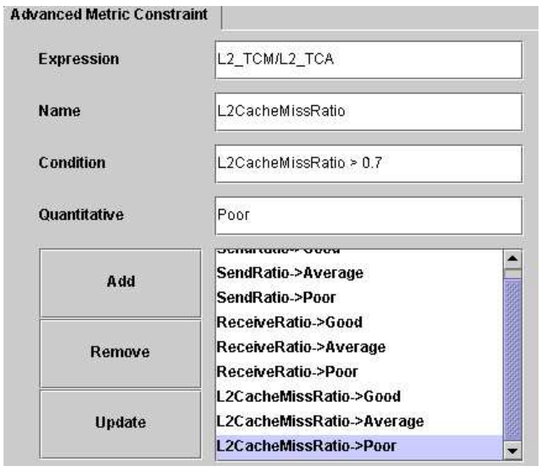
Clauses of Conditions

```
((TimingMetrics.name='wtime') AND (TimingMetrics.value > 5000000))
 OR ((TemporalOverheadMetrics.name='odata_recv') AND (TemporalOverheadMetrics.value > 3000000))
```

Processing Unit

| Computational Nodes | Processes | Threads |
|---|---|---|
| gsr402 | 0 | 0 |
| gsr404 | 1 | |

**Fig. 5.7.** Interface for search and filter of performance data in SCALEA

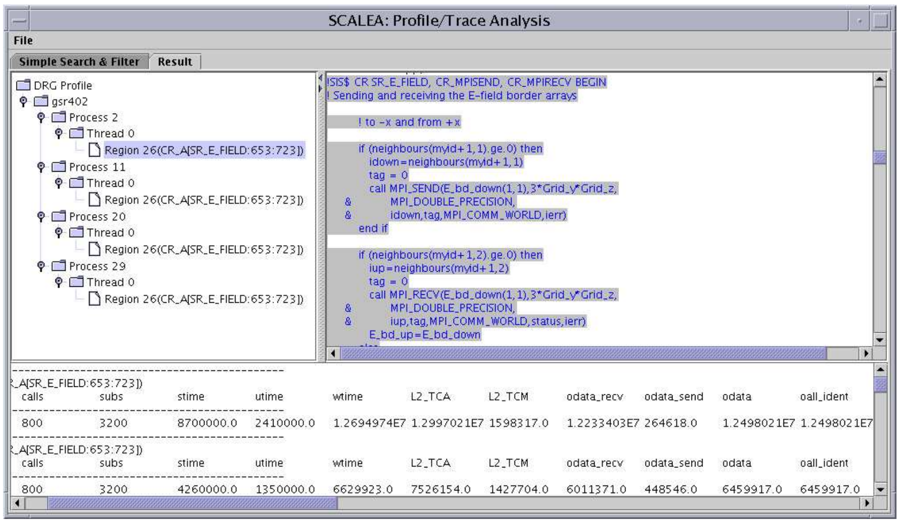


| | |
|---|---|
| Expression | L2_TCM/L2_TCA |
| Name | L2CacheMissRatio |
| Condition | L2CacheMissRatio > 0.7 |
| Quantitative | Poor |

SendRatio->Good
SendRatio->Average
SendRatio->Poor
ReceiveRatio->Good
ReceiveRatio->Average
ReceiveRatio->Poor
L2CacheMissRatio->Good
L2CacheMissRatio->Average
L2CacheMissRatio->Poor

**Fig. 5.8.** Specify complex performance conditions.

Figure 5.7 presents the interface for search and filter in SCALEA. The user can select any experiment for searching code regions under selection criteria. For each experiment, the user can choose *code region types* (e.g. send/receive,OpenMP loop), specify *metric constraints* based on performance metrics (timing, hardware parameter, overhead), and opt the *processing unit*



```
DRG Profile
  gsr402
    Process 2
      Thread 0
        Region 26(CR_A[SR_E_FIELD:653:723])
    Process 11
      Thread 0
        Region 26(CR_A[SR_E_FIELD:653:723])
    Process 20
      Thread 0
        Region 26(CR_A[SR_E_FIELD:653:723])
    Process 29
      Thread 0
        Region 26(CR_A[SR_E_FIELD:653:723])
```

```
!SIS$ CR SR_E_FIELD, CR_MPISEND, CR_MPIRECV BEGIN
! Sending and receiving the E-field border arrays

      ! to -x and from +x

      if (neighbours(myid+1,1).ge.0) then
        idown=neighbours(myid+1,1)
        tag = 0
        call MPI_SEND(E_bd_down(1,1),3*Grid_y*Grid_z,
   &          MPI_DOUBLE_PRECISION,
   &          idown,tag,MPI_COMM_WORLD,ierr)
      end if

      if (neighbours(myid+1,2).ge.0) then
        iup=neighbours(myid+1,2)
        tag = 0
        call MPI_RECV(E_bd_down(1,1),3*Grid_y*Grid_z,
   &          MPI_DOUBLE_PRECISION,
   &          iup,tag,MPI_COMM_WORLD,status,ierr)
        E_bd_up=E_bd_down
```

```
---------------------------------------------
_A[SR_E_FIELD:653:723])
 calls    subs    stime       utime       wtime        L2_TCA       L2_TCM      odata_recv   odata_send  odata        oall_ident
---------------------------------------------
 800      3200    8700000.0   2410000.0   1.2694974E7  1.2997021E7  1598317.0   1.2233403E7  264618.0    1.2498021E7  1.2498021E7
---------------------------------------------
_A[SR_E_FIELD:653:723])
 calls    subs    stime       utime       wtime        L2_TCA       L2_TCM      odata_recv   odata_send  odata        oall_ident
---------------------------------------------
 800      3200    4260000.0   1350000.0   6629923.0    7526154.0    1427704.0   6011371.0    448546.0    6459917.0    6459917.0
```

**Fig. 5.9.** Results of performance search.

(computational nodes, processes, threads) on which the code regions are executed. Metric constraints can be defined in simple way such as defining ranges of values of performance metrics. They also can be constructed by selecting quantitative characteristics (see Figure 5.8). For instance, the user may define constraints for L2 cache miss ratio by expressing the L2 cache miss ratio as

$$L2CacheMissRatio = \frac{L2\_TCM}{L2\_TCA} \tag{5.10}$$

and then discretizing the L2 cache miss ratio:

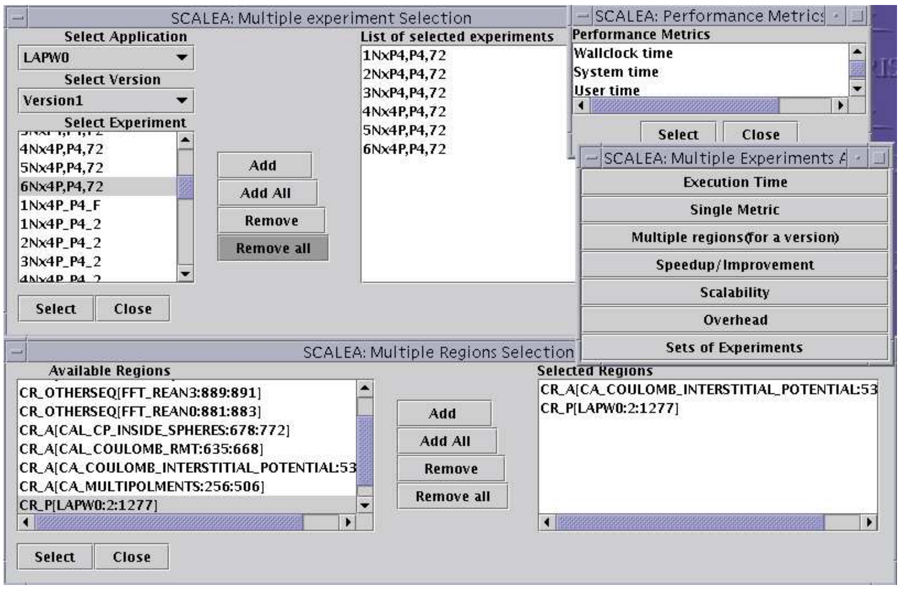
- *good* if $L2CacheMissRatio \leq 0.3$
- *average* if $0.3 < L2CacheMissRatio \leq 0.7$, and
- *poor* if $L2CacheMissRatio > 0.7$.

These quantitative characteristics can be stored into the experiment data repository for later use.

User-specified conditions will be transfered into queries in SQL language. SCALEA uses these queries to perform the search. For example, the result of the above-mentioned example is shown in Fig. 5.9. In the top-left window, the list of resulting code regions fulfilled the search conditions is visualized. By clicking into a code region, the source code and corresponding performance metrics of the code region will be shown in the top-right and bottom window, respectively.

#### 5.5.3.2 Multi-Experiment Analysis

Most existing performance tools [172, 167, 179, 204, 258, 82] investigate the performance for individual experiments one at a time. To employ the repository to archive data of experiments, SCALEA goes beyond this limitation by supporting multiple experiments. The user can select several experiments, code regions and performance metrics of interest whose associated data are stored in the data repository (see Figure 5.10). The outcome of every selected code region and metric is then analyzed and visualized for all experiments.


SCALEA: Multiple experiment Selection

Select Application: LAPW0

Select Version: Version1

Select Experiment:
4Nx4P,P4,72
5Nx4P,P4,72
6Nx4P,P4,72
1Nx4P_P4_F
1Nx4P_P4_2
2Nx4P_P4_2
3Nx4P_P4_2

List of selected experiments:
1NxP4,P4,72
2NxP4,P4,72
3NxP4,P4,72
4Nx4P,P4,72
5Nx4P,P4,72
6Nx4P,P4,72

SCALEA: Performance Metrics

Performance Metrics:
Wallclock time
System time
User time

SCALEA: Multiple Experiments A

SCALEA: Multiple Regions Selection

Available Regions:
CR_OTHERSEQ[FFT_REAN3:889:891]
CR_OTHERSEQ[FFT_REAN0:881:883]
CR_A[CAL_CP_INSIDE_SPHERES:678:772]
CR_A[CAL_COULOMB_RMT:635:668]
CR_A[CA_COULOMB_INTERSTITIAL_POTENTIAL:53
CR_A[CA_MULTIPOLMENTS:256:506]
CR_P[LAPW0:2:1277]

Selected Regions:
CR_A[CA_COULOMB_INTERSTITIAL_POTENTIAL:53
CR_P[LAPW0:2:1277]

**Fig. 5.10.** Multiple experiment analysis GUI.

Our multi-experiment analysis supports:

- *performance comparison for different sets of experiments*: The user can analyze the overall execution of the application across different sets of experiments; experiments are grouped based on their properties (e.g. problem sizes, communication libraries, platforms).
- *overhead analysis for multiple experiments*: Various sources of performance overheads across experiments can be examined.
- *study parallel speedup and parallel efficiency at both program and code region level*: Commonly, these performance metrics are investigated only at the level of the entire program. SCALEA, however, supports to examine the scalability at two levels: program and code region level.

We believe that multi-experiment analysis feature is very useful for scalability analysis of individual metrics and code regions for changing problem and machine sizes and underlying computing resources.

#### 5.5.3.3 Data Sharing and Tool Integration

A key reason for utilizing the data repository is the need to support data sharing and tool integration. With the complexity of current applications and architectures, it is unlikely that a single tool can provide enough functionality to fulfill the user's need. Data sharing and tool integration will allow the user to utilize facilities provided by different tools with available performance data. Data collected and analyzed by SCALEA is stored into the experiment data repository. Via well-defined interfaces, other tools can retrieve data from the experiment data repository and perform their own analysis.

In the framework of ASKALON, SCALEA provides basic components which are utilized by other tools. For example, performance metrics of code regions provided by SCALEA is used to compute the performance properties [83]. AKSUM [87] which is a high-level semi-automatic performance bottleneck analysis depends on SCALEA for instrumentation and measurement of code regions, and for analysis of performance overheads of user's programs. Based on performance metrics (overheads, timing, hardware parameters), AKSUM computes performance properties and conducts the high-level bottleneck analysis by performing the search for performance bottlenecks by evaluating these properties.

The PerformanceProphet [195] which supports for performance modeling and prediction on cluster architectures uses measured data stored in the experiment data repository to build the cost function for the application model which is represented in UML forms. The model is then simulated in order to predict the execution behavior of parallel programs.

ZENTURIO [201] is a tool purposely designed to support the management and control of performance experiments, parameter studies, and software testing. ZENTURIO uses the ZEN macro language (based on directives) to specify a possibly large set of interesting experiments for an application. ZENTURIO depends on SCALEA for automatically instrumentation of application sources. Thereafter, these sources are compiled and executed on target machines. After the experiments have been completed, SCALEA stores performance data into the experiment data repository. By using performance results analyzed by SCALEA, ZENTURIO visualizes performance data along with output data across multiple experiments.

Through integrating tools and leveraging performance data, various tools will provide flexible ways for the user to conduct performance analysis. The user can start with one analysis. If this analysis is not enough for his purpose, the user can launch other (high-level) analyses.

## 5.6 A Soft Computing-based Approach to Performance Analysis of Parallel and Distributed Programs

The development of high performance computers and the advances in communication have fostered the development of large-scale and complex applications. When tuning and optimizing the performance of such large-scale and complex applications, a large amount of performance measurements collected needs to be analyzed. This results in many challenges to be solved by current performance analysis tools for parallel and distributed programs because most of them support performance analysis via interactive statistical graphics which do not work well with large amount of data. Performance analysis tools have to exploit and develop new scalable and intelligent methods to examine that huge amount of performance measurements.

In response to these challenges, recently performance analysis community has focused on developing performance tools for parallel and distributed programs that are capable of supporting automatic performance analysis [261, 268, 87], dealing with large performance data sets [22], and analyzing multiple experiments [246]. However the development of automatic and intelligent performance analysis is still at an early stage. Currently, existing performance analysis tools use *exact* analysis methods that result in exact conclusions about performance characteristics of applications. Also these tools present the user with the performance data in the form of precise and complex numerical values. However, performance measurements collected may have limited accuracy or missing data, in other words, performance measurements may be incomplete and uncertain data. Therefore, exact analysis methods might not be appropriate for processing uncertainty data. Secondly, performance tools based on numerical analysis interact with the user through complex numerical values which are not easily understood by human beings. Moreover, in real world we largely rely on domain expertise and user-provided inputs as parameters to control the performance analysis and tuning. Such expertise and inputs may be inexact and uncertainty. However, there is no mechanism to specify and control approximate and inexact parameters in existing performance analysis tools, in other words, these tools do not provide a mechanism for making soft decisions.

We present a new approach to automatic performance analysis that we call the *soft performance analysis*. In this approach, soft computing techniques such as fuzzy logic (FL), machine learning (ML) concept, and the combination of FL and ML are studied and developed for performance analysis of parallel and distributed programs. We use FL to represent performance characteristics and introduce the concepts of performance score and performance similarity based on fuzzy logic and similarity theory. Employing these concepts, we develop several techniques and methods for performance analysis such as fuzzy-based performance classification, ranking analysis and bottleneck search, and similarity analysis for single- and multi-experiment. A fuzzy-based query language is proposed that enables the search of performance data by using linguistic expressions, not complex numerical expressions as usual. Furthermore, we develop and implement a fuzzy C-means clustering for classifying the performance of code regions into classes of similar performance, and propose fuzzy rules for filtering performance data.

### 5.6.1 Hard Computing and Soft Computing

Hard computing applies exact methods which are based on binary logic, crisp system and numerical analysis. Thus hard computing requires a precise model. For example, if we measure the communication/computation ratio $T_{comm}/T_{comp}$ for a set of code regions $CR = \{r_1, r_2, \cdots, r_n\}$, we might want to ask *"does $r_i$ have* **high** $T_{comm}/T_{comp}$*?"*. With hard computing, we can use Boolean logic to define a condition as follows: any code region whose $T_{comm}/T_{comp} \geq 0.7$ has *high* $T_{comm}/T_{comp}$. However, that condition is very exact, for example why $T_{comm}/T_{comp} = 0.7$ is high and $T_{comm}/T_{comp} = 0.6999$ is not? In real world such condition is not exact but approximate. Also the knowledge used to define the condition seems to be uncertainty. Therefore, with hard computing, we may need a lot of computation in order to find the suitable solution for a problem.

Unlike hard computing, soft computing [278, 45, 202] is *"tolerant of imprecision, uncertainty, partial truth"* and approximation [278]. With the example above, we can represent *high* $T_{comm}/T_{comp}$ as a subset and a code region $r_i$ can be associated with a value which indicates the degree of membership of its $T_{comm}/T_{comp}$ value in subset *high*. For example, Figure 5.11 illustrates the function used to determine the degree of $T_{comm}/T_{comp}$ value in the subset *high*. Hence, with $T_{comm}/T_{comp} = 0.6999$ we know that the $T_{comm}/T_{comp}$ of the code region is *almost high*.

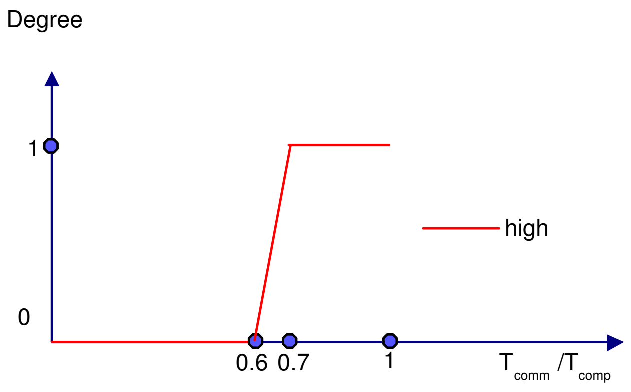


**Fig. 5.11.** Example of representing high $T_{comm}/T_{comp}$.

The main objective of soft computing is to *"exploit the tolerance for imprecision, uncertainty, partial truth, and approximation to achieve tractability, robustness and low solution cost"* [278, 45]. The basic principles underlying soft computing are inspired by fuzzy sets [273], the analysis of complex systems and decision processes [274], possibility theory and soft data analysis

[275]. The main techniques and tools of soft computing that we are exploiting are fuzzy logic (FL) and machine learning (ML).

### 5.6.2 Soft Performance Analysis

The current techniques in existing performance analysis tools are based on *hard* computing that is based on binary logic, crisp system and numerical analysis. For example, to classify the performance of parallel programs into performance classes performance analysis tools normally use a *characteristic function.* That is, given a performance metric and a set of *performance characteristic terms*, e.g. *poor, medium* and *good*, each term is associated with a data set, the performance of a code region is mapped into a member of the set of performance characteristic terms with respect to the given metric. Such classification is exact, e.g., with respect to the given metric, the code region is classified as *good* or *poor.* The characteristic function $\mu(x)$ can be represented as follows:

$$\mu(x) : X \to \{0, 1\} \tag{5.11}$$

where $X$ is the data set associated with a performance characteristic term and $x$ is the value of a performance metric. There is no common way to define $\mu(x)$ and $X$ which are dependent on specific analyses for specific applications. For example, in our previous work, presented in Section 5.5.3.1, $\mu(x)$ and $X$ are manually defined by users whereas in [261], when building the decision tree, they are automatically defined based on training data.

Currently, performance analysis tools are based on numerical analysis in which imprecision and uncertainty are not accepted. Since approximate search, classification and reasoning are not possible, the cycle of finding performance patterns in a large set of performance data has been lengthened because, in real world, various factors, e.g., the boundaries between performance classes, the performance search constraints, etc., might be fuzzy. These fuzzy factors possibly make exact methods not yield the expected results. Moreover, current tools focus on supporting the performance analysis through statistical graphics which are not well suited for processing large performance datasets.

In real world, performance data may be uncertainty and expertise used in performance analysis domain can be imprecision and uncertainty. For example, in the case of performance classification, the performance of a code region is classified into *good* but we do not know how grade of *good* is? It is possible that the performance of the code region is *a little good*, *fairly good* or *very good.* In many cases, the performance measurements collected are imprecision and uncertainty because we have limited measured instruments, imperfect performance models, etc. Because we may not be certain of performance data and expertise, we may accept some degree of tolerance about imprecision, uncertainty and approximate in our analysis techniques. Moreover, performance analysis tools have to replace large and complex numerical performance data which cannot be grasped by the end user by lower dimensional and summarized information, notations and concepts that can be easily understood by the end user.

To address the above-mentioned issues, we investigate the performance analysis techniques that are based on soft computing model. FL and ML, widely used in soft computing, are useful tools for processing and analyzing uncertainty and large-volume data. The soft performance analysis we propose aims to develop techniques for performance tools that can (i) extract useful performance information from large, dynamic and multi-relational performance measurement sources, (ii) support the specification and control of approximate and inexact parameters, commands and requests in existing performance analysis tools, and (iii) interact with the user through high level notions and concepts expressed in linguistic expressions. Soft computing plays a major role in achieving that aim.

We outline the soft performance analysis approach as follows. Firstly, FL theory can be used to represent and normalize performance data and to reduce the data volume. We can represent a *grade* (score) of a metric value by using a fuzzy set. Let $x$ be a metric value in the universe of discourse $U$. The grade of performance of $x$ in $U$ is measured by an extension of the characteristic function called *membership function* as follows:

$$\mu(x) : U \rightarrow [0, 1] \tag{5.12}$$

Here the membership function maps each $x \in U$ into a real number in the range from 0 to 1. That is

$$0 \leq \mu(x) \leq 1 \tag{5.13}$$

with 0 meaning that $x$ has the lowest grade, 1 meaning that $x$ has the highest grade.

The application of fuzzy logic theory also involves the concept of linguistic variables. The use of linguistic variables is particular useful to the end-user as humans employ mostly words in computing. This has been realized by the concept of computing with words [277]. By using fuzzy logic, performance tools can provide a way to perform analysis and to interpret performance results with linguistic terms that is close to the way in which humans interpret linguistic variables.

Secondly, when processing large and diverse performance data, information about performance summaries, similarities and differences among data items become more important as we cannot examine each data items in detail. Similarity measure techniques can be exploited to reveal the performance similarities and differences. Machine learning techniques [174] which are widely used in data mining [121] can be utilized to discover patterns in very large performance datasets. For example, machine learning is combined with fuzzy computing to provide fuzzy clustering of performance data that can be used to classify the multi-dimensional performance measurements into relatively small groups of similar objects.

### 5.6.3 Representing Performance Characteristics with Fuzzy Logic

In this section, we outline the representation of performance characteristics by using concepts of fuzzy logic such as *fuzzy set, membership function*, and *modifier*. An outline of fuzzy sets, common membership functions and modifiers can be found in Appendix F.

A fuzzy set $FS$ is used to map metric values onto membership values that lie in the range $[0, 1]$. An $FS$ is expressed as a set of ordered pairs $FS = \{(x, \mu(x))|x \in U\}$ where $\mu(x)$ is the membership function that gives the degree of membership of $x$ in $U$, and $U$ is the universe of discourse of $x$. Each fuzzy set is represented by a performance characteristic term.

Let $v$ be a metric value with the universal of discourse $U$. $U$ is characterized by a given set of performance characteristic terms $T = \{t_1, t_2, \cdots, t_n\}$; performance characteristic terms are linguistic terms such as *poor, medium* and *high*. Each characteristic term $t_i$ is associated with a membership function $\mu_i(x)$ which determines the membership of $x$ in $t_i$. $v$ can be classified according to these terms.

A *modifier* is used to enhance the use of performance characteristic terms. A modifier (e.g. *very* or *slightly*) is an operation that modifies a performance characteristic term (e.g. *poor* or *bottleneck*) (this also means to modify the shape of the fuzzy set representing the term). The modification results in a new fuzzy set represented by a new phrase (e.g., *very poor* or *slightly bottleneck*): the new fuzzy set fits the meaning of the new phrase. These modifiers are also commonly referred to as *hedges*.

### 5.6.4 Performance Score

When evaluating and comparing performance of code regions most existing performance tools use quantitative measurement values and do not employ quantization or normalization techniques to evaluate the performance based on multiple metrics. As a result, it is difficult to evaluate and compare the performance of code regions with multiple metrics whose scales and data ranges are different.

We present the concept of *performance score* which is used to evaluate the performance of a code region within a base, e.g. the parent code region or the whole program. The concept is based on (i) a set of selected performance metrics representing the performance of the code region, and (ii) a weight set representing the significance of performance metrics. Given a code region $r$, let $rs$ be the region summary of $r$ with a set of $n$ performance metrics $\{m_1, m_2, \cdots, m_n\}$. Suppose the number of performance metrics measured is the same for every code regions. The region summary $rs$ can be represented in $n$ dimensional space. Let $v_i = rs(m_i)$ be the value of metric $m_i$ in region summary $rs$ and let $s_i$ be a score that represents the performance of region summary $rs$ with respect to metric $m_i$. We compute $s_i$ as follows

$$s_i = \mu(v_i), \mu(x) : [0, V_{m_i}] \rightarrow [0, 1] \tag{5.14}$$

where $\mu(x)$ is the membership function that determines the performance score, and $V_{m_i}$ is the maximum value of metric $m_i$ observed. The observed $V_{m_i}$ is dependent on the level of code region analysis. For example, if we analyze performance scores of $rs$ with its parent $rs_{parent}$ as the base, $V_{m_i} = rs_{parent}(m_i)$. However, if the entire program is selected as the base, $V_{m_i} = rs_{program}(m_i)$ where $rs_{program}$ is the region summary of the code region representing the entire program.

The value of $s_i$ is in the range $[0, 1]$ with 0 means the lowest score, 1 means the highest score. A higher performance score might imply a higher performance (*higher is better* class, see Section 3.7) but it may be used to indicate a lower performance (*lower is better* class, see Section 3.7). The exact semantics of the value of the performance score is defined by specific implementations. As a result, performance scores can be used in various contexts such as

- to represent a significant impact level: the higher a performance score is, the higher impact the code region has
- to represent a severity: the higher a performance score is, the more severe the core region is.

There are several ways to select $\mu(x)$, depending on the specific analysis and approximate model used. The most simple way is to define the membership function $\mu$ as $\mu(v_i) = \frac{v_i}{V_{m_i}}$ which assumes that the score is based on linear model. For performance metrics with non-linear model we can choose trapezoid, S-function, Z-function, triangle, or user-defined function as $\mu(x)$ (see Appendix F for common membership functions).

Each region summary $rs$ is associated with a vector of performance scores $\vec{s}$. However, depending on the purpose of the performance analysis, we may only select a subset of performance scores of the vector as metrics for analyzing the performance of the code region. Like quantitative measurement values, we can compare two performance scores of two different metrics. However, as performance scores are normalized values, we can aggregate performance scores $\vec{s}$ of a region summary into a single score by using the overall weighted average (OWA) operator.

Given a set of performance metrics $M = \{m_1, m_2, \cdots, m_n\}$, let $\{s_1, s_2, \cdots, s_n\}$ be performance scores of $rs$. Let $W = \{w_1, w_2, \cdots, w_n\}$ be the set of weights; $w_i$ is a weight factor associated with metric $m_i$. The aggregation performance score for a score set $\vec{s}$, $OWA(\vec{s})$, may be computed as follows

$$OWA(\vec{s}) = \frac{\sum_{i=1}^{n}(|s_i w_i|)}{\sum_{i=1}^{n} w_i} \tag{5.15}$$

For the sake of simplicity, normally $w_i \in (0, 1)$ and $\sum_{i=1}^{n} w_i = 1$. OWA score is particular useful for support of decision making in performance analysis and tuning because very often in performance analysis we have to decide *where* and *what* to optimize: which are the focused metrics of which code regions that should be tuned and optimized in order to achieve a better performance. With code regions having `odata_send` or `odata_recv` or both, we optimize the performance of the code region having highest `odata_send` or `odata_recv` score or we want to balance between the two. OWA allows specifying our preferences. Hence we use the notation $(m_i, w_i)$ to denote a metric $m_i$ with its associated weight $w_i$, for example `(odata_send,0.75)` means metric `odata_send` with weight 0.75.

We can use a single performance score or a set of performance scores with or without weight set as metrics in evaluating the performance of code regions. Using OWA we can provide a single metric representing for multiple performance metrics. Not only that provides comfortable methods to normalize and characterize the performance metrics of code regions but also it reduces the computation as data has been transformed from n-dimensional data points to a small set of performance scores. Table 5.8 presents an example in which we use different membership functions and weight sets to compute performance scores for a set of selected code regions in an experiment. The resulting performance scores are dependent on the selection of membership functions, performance metrics and weight set. As performance data and model contain sources of uncertainty, we can try to use various membership functions to compute the performance score under various uncertain conditions.

| **Code Region** | **Linear {(wtime,1)}** | **S-function {(wtime,1)}** | **Linear {(utime,0.5), (stime,0.5)}** | **S-function {(utime,0.5), (stime,0.5)}** |
|---|---|---|---|---|
| **CAL_POWER** | 0.014775 | 0.00369 | 0.0522 | 0.013 |
| **IONIZE_MOVE** | 0.48712 | 0.4774 | 0.269 | 0.274 |
| **SR_E_FIELD** | 0.01362 | 0.0034 | 0.0677 | 0.018 |
| **PARTICLE_LOAD** | 1.0325E-4 | 2.581E-5 | 4.543E-5 | 1.135E-5 |

**Table 5.8.** Example of performance scores with various membership functions, performance metrics and weight sets. The performance scores of code regions are computed relatively to the whole program.

#### 5.6.4.1 Performance Scores versus Performance Properties

APART working group proposes performance properties which can be seen as indices used to determine the severity of the performance [83]. The basic tenet of performance score and performance property is to *normalize* value of a performance metric into the range $[0, 1]$. However, the two concepts are different.

Firstly, the membership function of a performance score is not predefined and fixed during the analysis. Rather, the membership function should be selected by the performance tools, maybe under user preferences, during the analysis. In contrast, a performance property is computed by a fixed and predefined function. Performance severity is computed based on single metric whereas performance of a code region normally is characterized by a set of performance metrics. In addition, the performance severity is computed based on linear model which normally is not suitable for all performance metrics. Another aspect is the utility of a performance metric. Performance severities are metrics that belong *lower is better* class (see Section 3.7) only. However, a performance tool needs to provide a wide range of performance metrics, which belong to different utility classes, because the user is not always interested in examining a single class of utility function.

A performance property cannot represent a set of performance metrics. Instead, performance properties can be grouped into a collection, as discussed in [87]. However, there is no concept of weight operator associated with performance properties. A set of performance properties is just a collection with simple operators, e.g. min and max. Our performance score is based on theory of fuzzy logic. Fuzzy logic also allows the representation of fuzzy concepts, such as *near* and *very*, which cannot be found in performance properties. In fact, performance property can be considered a special case of performance score.

### 5.6.5 Performance Similarity

Most existing performance tools examine executions of code regions by visualizing performance measurements on numerous displays including process time-lines with zooming and scrolling, histograms of state durations and message data. Those displays are crucial but they do not measure the similarity among executions, not to mention that they mostly require all data to be loaded into the memory, eventually making the tools in-scalable. The user has to observe the displays and perceive the similarity and the difference among these values. Moreover, it is difficult to compare multivariate data through visualization. Similarity measure can help uncovering *similar performance patterns* among code regions. That performance patterns are not easily perceived through complex numerical numbers and displays, or by examining a large data set. Performance similarity can be measured for executions of code regions in a single experiment or across multiple experiments. By measuring similarity between executions of a code region, we can examine whether the performance pattern of a code region is regular or not when the code region is executed on different processing units of experiment(s). Analyzing the similarity among different versions of a code region, e.g., different implementations of a function, helps to discover the similarity between their performance patterns.

We propose methods to compute the *performance similarity measure* which can be used as a metric to indicate the performance similarity among executions of code regions and among experiment factors. Intuitively, a similarity measure between two region summaries $rs_i$ and $rs_j$ can be defined as a function which maps region summaries presented by performance score vectors into numbers which indicate the degree of similarity between the performance of code regions. Formally, let $o_i$ and $o_j$ be objects (e.g. region summaries, experiment factors), a similarity measure is a function $sim(o_i, o_j) \to [0, 1]$ that compares $o_i$ with $o_j$. The result of $sim$ is in the range of $[0, 1]$ where 0 denotes complete dissimilarity and 1 denotes complete similarity[3].

#### 5.6.5.1 Performance Similarity Measure for Code Regions

Let $rs_i$ and $rs_j$ be region summaries of $r$ in experiment $e$, respectively. Let $s_{il}$ and $s_{jl}$ be performance scores of $rs_i$ and $rs_j$ with respect to metric $m_l$, respectively. We use Equation 5.14 to compute $s_{il}$ and $s_{jl}$ as follows

$$s_{ij} = \mu_l(v_{il}), \mu_l(x) : [0, V_{m_l}] \to [0, 1] \tag{5.16}$$

$$s_{jl} = \mu_l(v_{jl}), \mu_l(x) : [0, V_{m_l}] \to [0, 1] \tag{5.17}$$

where $V_{m_l}$ is the maximum observed value of metric $m_l$. $V_{m_l}$ depends on the base on which the similarity analysis is conducted. For example, if we measure similarity of $rs_i$ and $rs_j$ with their parent $rs_{parent}$ as the base then $V_{m_l} = rs_{parent}(m_l)$.

We then define *similarity measure* $sim_{ij}$ between two region summaries $rs_i, rs_j$ as follows

$$d_{ij} = \sqrt{\sum_{l=1}^{n} (|s_{il} - s_{jl}|^2 w_l)} \tag{5.18}$$

$$sim_{ij}(rs_i, rs_j) = 1 - d_{ij} \tag{5.19}$$

where $w_l$ is a weight factor for metric $m_l$. $d_{ij}$ is the distance measure between $rs_i$ and $rs_j$ that is computed by using Euclidean function. Note that a variety of distance functions is available such as Minkowski, Manhattan, Correlation and Chi-square. Instead of using performance scores as metrics used to compute the distance, we can also use measurement data. However, as measurement data have different scales (non-normalized values), the resulting distance measure can significantly be impacted.

To determine the performance similarity among executions of code regions across a set of experiments, we also use Equation 5.19 to measure the performance similarity. However, given a performance metric $m_i$, when determining performance score the maximum observed value $V_{m_i}$ is the maximum value obtained from a base experiment (e.g. the experiment with the highest execution time, the experiment of sequential version). Given a code region $r$ and a set of experiments $\{e_1, e_2, \cdots, e_n\}$. Let $rs_i$ be region summary of $r$ in experiment $e_i$. We compute similarity measure $sim(rs_1, rs_i)$, $i : 2 \to n$ by using various membership and distance functions.

[3] Although the similarity measure and fuzzy membership degree are in the same range $[0, 1]$, they are based on two different concepts. The similarity measure may be computed based on fuzzy metrics, but not necessarily.

#### 5.6.5.2 Performance Similarity Measure for Experiment Factors

The similarity analysis presented in Section 5.6.5.1 allows us to examine the similarity between executions of code regions across a set of experiments, in other words, it provides similarity between the performance of code regions of experiments. It, however, does not take into account the similarity among factors of experiments. Experiment factors which can be controllable, e.g. problem size, number of CPUs and communication libraries, or uncontrollable, such as system load and OS processes, have significant impact on the performance of the applications. Without considering similarity between experiment factors, it is difficult to explain cases in which the performance of code regions is not similar because the experiment factors can be different. Therefore, initially we try to address this problem by measuring similarity between controllable factors.

Let $sim_f(e_i, e_j)$ be similarity measure for factor $f$ between experiments $e_i$ and $e_j$. Given a set of controllable factors $F = \{f_1, f_2, \cdots, f_n\}$, similarity measure is computed for each factor $f_i \in F$. There is no common way to compute $sim_f$ as a controllable factor and its role are dependent on each experiment and are different from experiment to experiment. The objective of our analysis is to find out the relationship between the similarity of the performance of code regions (outputs of performance experiments), $sim_o$ (e.g. $sim(rs_i, rs_j)$), and $sim_{f_i}$. Assume we know the similarity degree between controllable factors and between the performance of code regions, we can establish the model among them. Naturally we expect the similarity measures of the controllable factors of two experiments and of the performance of these experiments behave in a similar fashion, e.g. if the controllable factors are very similar then the performance of experiments should be very similar.

### 5.6.6 Fuzzy-based Performance Classification

Performance classification uses prior knowledge to classify performance of code regions according to performance characteristic terms. Formally, given a metric value $v$ and a set of performance characteristic terms $T = \{t_1, t_2, \cdots, t_n\}$, $v$ are classified according to that terms. In existing performance tools, the classification gives exact result, that is, $v$ belongs to only one $t_i \in T$, with no degree of membership, e.g. in [261]. The advantage of fuzzy-based performance classification is that it provides a soft decision by giving a value that describes the degree to which a code region fits into a performance class, rather than only a hard decision which indicates whether a code region belongs to a performance class or not.

To classify performance of code regions, we firstly define membership functions for performance metric $m$ by partitioning the range of the value of metric $m$ into segments. Each segment is described by a performance characteristic term which is associated with a fuzzy set. This process can be done automatically. Once membership functions are determined, the fuzzification that takes quantitative value $v$ of $m$ and determines membership degree of $v$ to membership functions is conducted.

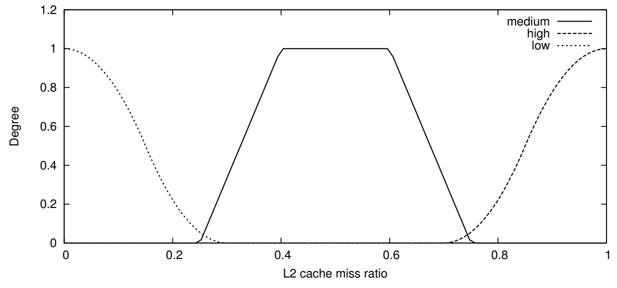
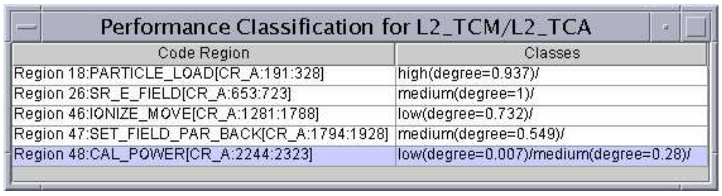


**Fig. 5.12.** Performance characteristic terms $T = \{low, medium, high\}$ with their associated fuzzy sets.

To demonstrate this analysis, we classify code regions of 3DPIC application (see Section 7.3.4 for more information about 3DPIC) executed with 4 processes according to a set of performance characteristic terms $T = \{low, medium, high\}$ representing the L2 cache miss ratio. Three different fuzzy sets Z-function, trapezoid and S-funtion are associtated with *low, medium, high* term, respectively, as shown in Figure 5.12. We then conduct the classification for some selected code regions. Figure 5.13 presents the result with five selected code regions. As shown in Figure 5.13, the code region `PARTICLE_LOAD` is *high* L2 cache miss ratio. However, code region `CAL_POWER` is member of both *low* and *medium*.



Performance Classification for L2_TCM/L2_TCA

| Code Region | Classes |
|---|---|
| Region 18:PARTICLE_LOAD[CR_A:191:328] | high(degree=0.937)/ |
| Region 26:SR_E_FIELD[CR_A:653:723] | medium(degree=1)/ |
| Region 46:IONIZE_MOVE[CR_A:1281:1788] | low(degree=0.732)/ |
| Region 47:SET_FIELD_PAR_BACK[CR_A:1794:1928] | medium(degree=0.549)/ |
| Region 48:CAL_POWER[CR_A:2244:2323] | low(degree=0.007)/medium(degree=0.28)/ |

**Fig. 5.13.** Membership in {*low, medium, high*} L2 cache miss ratio for selected code regions of 3DPIC.

We can also build new characteristic terms by combining existing ones with modifiers. For example, we can classify code regions according to *very low* L2 cache miss ratio; the term *very* is a fuzzy modifier. Let $\mu_{low}(x)$ be the fuzzy set associated with term *low* and $\mu_{very-low}(x)$ be the fuzzy set associated with term *very low*. Formally $\mu_{very-low}(x)$ is a function that map $\mu_{low}(x)$ onto $[0, 1]$

$$\mu_{very-low}(x) = f(\mu_{low}(x)), f \in [0, 1] \tag{5.20}$$

In practice, the term *very* is often approximated by $f(x)^2$, for example $\mu_{very-low}(x) = (\mu_{low}(x))^2$. The use of modifiers allows us to extend the set of characteristic terms, enhancing the ability to describe performance characteristic terms.

In the current prototype, as many available fuzzy sets can be used to represent a term, we base on the most popular fuzzy sets for describing membership of continuous data such as triangle, trapezoid, S-function and Z-function (see Appendix F).

### 5.6.7 Fuzzy-based Query for Searching Performance Data

Most existing performance analysis tools lack a query language that can be used to search performance data. A search language that is in the form of human notions or closer to what people use to think is a useful tool to support the end users in conducting performance analysis.

The fuzzy-based approach offers the possibility of searching performance data with words. Fuzzy-based search by using linguistic expressions has been widely employed in database systems, information retrieval, etc., [48, 206, 67]. For searching performance data with words, a fuzzy-based query language can be built based on performance characteristic terms along with a set of fuzzy modifiers, and *AND* and *OR* operators.

```
⟨PERFQL_Statement⟩ ::= ⟨PERFQL_Expr⟩ | ⟨PERFQL_Statement⟩
                       OR ⟨PERFQL_Expr⟩
⟨PERFQL_Expr⟩      ::= ⟨PERFQL_Term⟩ | ⟨PERFQL_Expr⟩
                       AND ⟨PERFQL_Term⟩
⟨PERFQL_Term⟩      ::= (⟨METRIC⟩ is ⟨F_Expr⟩)
⟨F_Expr⟩           ::= ⟨F_Term⟩ | ⟨F_Expr⟩ OR ⟨F_Term⟩
⟨F_Term⟩           ::= ⟨M_Expr⟩ | ⟨F_Term⟩ AND ⟨M_Expr⟩
⟨M_Expr⟩           ::= MODIFIER ⟨M_Expr⟩ | ⟨P_Term⟩
⟨P_Term⟩           ::= PERF-TERM | ⟨(F_Expr)⟩
```

**Fig. 5.14.** BNF description of the top-level syntax of PERFQL.

We propose a fuzzy-based query language for searching performance data. Figure 5.14 presents our PERFQL (performance query language based on fuzzy logic). *METRIC* is a metric name or a *metric expression.* A metric expression consists of operands and *+, -, *, /* arithmetic operators; operands are metric names. `PERF-TERM` is a performance characteristic term, each associates with a fuzzy set. `MODIFIER` is a modifier for performance characteristic terms. *F_Expr* describes the syntax of generic linguistic expressions representing performance characteristic terms[4].

For example, the following query can be used to find code regions which have high wall-clock time and poor L2 cache miss ratio

```
(wtime is HIGH_EXECUTION_TIME) AND (L2_TCM/L2_TCA is
 POOR_CACHE_MISS)
```

where `wtime` and `L2_TCM/L2_TCA` denote wall-clock time and L2 cache miss ratio, respectively. `HIGH_EXECUTION_TIME` and `POOR_CACHE_MISS` are performance characteristic terms, each is represented by a fuzzy set.

PERFQL allows the user to easily define queries for searching performance data by using words, not numerical expressions. Thus, it is easy to be understood and interpreted by the user. Moreover, fuzzy-based queries enable approximate search. When using crisp condition, performance data which is outside the selected range is totally excluded. However, in real world, performance data slightly less or greater than the crisp condition might be of interest as the selected range may not be the right choice. With soft approach, we can easily obtain that data by using modifiers along with performance characteristic terms.

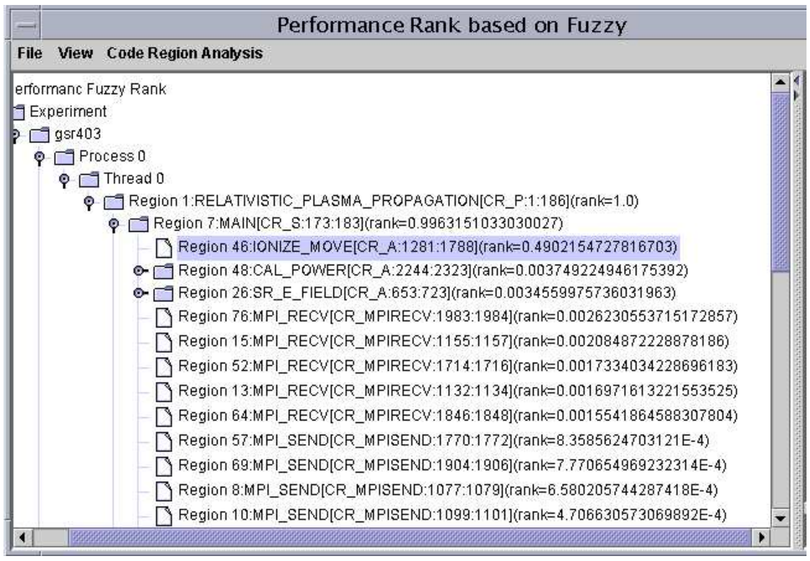
### 5.6.8 Fuzzy Ranking Analysis

Ranking analysis has been used widely in evaluating systems and resources. Most existing performance tools provide a simple type of ranking analysis in which performance of code regions is sorted based on measurement value of a selected metric. A performance property [83] can also be seen as a simple metric that can be used to rank the performance of code regions (the higher value of a performance property is, the more severe the code region is). However, all mentioned ranking schemes are based on single metric whereas the user may wish to rank the performance of code regions based on set of performance metrics, not on a single one. With ranking analysis that uses raw measurement value it is difficult to interpret and compare the significance of the performance of code regions. An index like performance property is mostly computed based on linear model.

We propose a fuzzy ranking algorithm, based on fuzzy logic and OWA operators, which can be used to rank the significant level of performance of code regions compared to another code region called the base. Let $RS = \{rs_1, rs_2, \cdots, rs_n\}$ be the set of region summaries to be ranked, $rs_i$ contains performance values of $n$ metrics $\{m_1, m_2, \cdots, m_n\}$. For $m_i$, we select a membership function $\mu_i$. The membership functions can be linear or non-linear,

[4] In [187, 101], a BNF (Backus Naur Form) that describes the syntax of generic fuzzy linguistic expressions (e.g. *very low or medium*) is introduced. PERFQL reuses that BNF. That BNF is equivalent to our BNF describing $\langle F\_Expr \rangle$ of PERFQL.

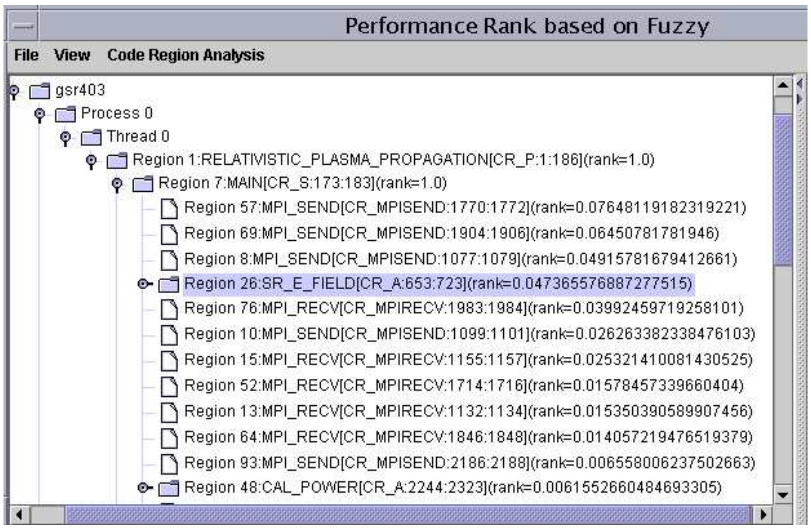
depending on the model of the performance metric. For each $rs_i$, by using membership functions, we measure performance score vector $\vec{s_i}$ with the parent code region or the whole program as the base. We then apply OWA operator to compute an OWA score from $\vec{s_i}$. This OWA score is used to rank code regions in an experiment. We apply our performance rank for various types of performance metrics such as measured timing, measured counter, ratio and overhead metrics.



```
erformanc Fuzzy Rank
Experiment
 gsr403
   Process 0
     Thread 0
       Region 1:RELATIVISTIC_PLASMA_PROPAGATION[CR_P:1:186](rank=1.0)
         Region 7:MAIN[CR_S:173:183](rank=0.9963151033030027)
           Region 46:IONIZE_MOVE[CR_A:1281:1788](rank=0.4902154727816703)
           Region 48:CAL_POWER[CR_A:2244:2323](rank=0.003749224946175392)
           Region 26:SR_E_FIELD[CR_A:653:723](rank=0.003455997573603196З)
           Region 76:MPI_RECV[CR_MPIRECV:1983:1984](rank=0.0026230553715172857)
           Region 15:MPI_RECV[CR_MPIRECV:1155:1157](rank=0.002084872228878186)
           Region 52:MPI_RECV[CR_MPIRECV:1714:1716](rank=0.001733403422869618З)
           Region 13:MPI_RECV[CR_MPIRECV:1132:1134](rank=0.0016971613221553525)
           Region 64:MPI_RECV[CR_MPIRECV:1846:1848](rank=0.0015541864588307804)
           Region 57:MPI_SEND[CR_MPISEND:1770:1772](rank=8.3585624703121E-4)
           Region 69:MPI_SEND[CR_MPISEND:1904:1906](rank=7.770654969232314E-4)
           Region 8:MPI_SEND[CR_MPISEND:1077:1079](rank=6.580205744287418E-4)
           Region 10:MPI_SEND[CR_MPISEND:1099:1101](rank=4.706630573069892E-4)
```

**Fig. 5.15.** Performance rank for 3DPIC, executed on 4 processors, using {`(wtime,1)`}.

Figure 5.15 presents the performance rank for 3DPIC application executed on 4 processors. Ranking is based on the performance score which indicates the significant impact of the performance, the higher value a score has, the higher significant impact it indicates. For each code region, we evaluate its rank with its parent code region as the base. For example, in Figure 5.15 the code region `MAIN` has rank about 0.99, suggesting it dominates performance of the program. Inside `MAIN`, `INONIZE_MOVE` is the top in performance rank of code regions of `MAIN` with the score approximate to 0.49. However, code region `CAL_POWER` has the score approximate to 0.003 even it is the second of the list. This suggests that most instrumented code regions have little impact on the total execution time. Figure 5.16 shows another example of using ranking analysis to investigate the severity of the overhead due to sending and receiving data (data movement overhead). The performance score indicates the overhead severity, the higher value a score has, the more severe overhead the code region has. Although `INONIZE_MOVE` has significant impact on the execution of `MAIN`, it does not contribute to data movement overhead. Also no code region has significant impact on the data movement overhead because all performance scores have small values. However, there are many code regions which are sources of data movement overhead.



```
gsr403
  Process 0
    Thread 0
      Region 1:RELATIVISTIC_PLASMA_PROPAGATION[CR_P:1:186](rank=1.0)
        Region 7:MAIN[CR_S:173:183](rank=1.0)
          Region 57:MPI_SEND[CR_MPISEND:1770:1772](rank=0.07648119182319221)
          Region 69:MPI_SEND[CR_MPISEND:1904:1906](rank=0.06450781781946)
          Region 8:MPI_SEND[CR_MPISEND:1077:1079](rank=0.04915781679412661)
          Region 26:SR_E_FIELD[CR_A:653:723](rank=0.047365576887277515)
          Region 76:MPI_RECV[CR_MPIRECV:1983:1984](rank=0.03992459719258101)
          Region 10:MPI_SEND[CR_MPISEND:1099:1101](rank=0.02626338233847610З)
          Region 15:MPI_RECV[CR_MPIRECV:1155:1157](rank=0.025321410081430525)
          Region 52:MPI_RECV[CR_MPIRECV:1714:1716](rank=0.01578457339660404)
          Region 13:MPI_RECV[CR_MPIRECV:1132:1134](rank=0.015350390589907456)
          Region 64:MPI_RECV[CR_MPIRECV:1846:1848](rank=0.014057219476519379)
          Region 93:MPI_SEND[CR_MPISEND:2186:2188](rank=0.006558006237502663)
          Region 48:CAL_POWER[CR_A:2244:2323](rank=0.0061552660484693305)
```

**Fig. 5.16.** Performance rank for 3DPIC, executed on 4 processors, using {`(odata_recv,0.5), (odata_recv, 0.5)`}.

With our fuzzy ranking analysis, the user can rank the performance of code regions based on multiple metrics and weight set. Ranking analysis can be customized to examine the performance from different angles, such as the level of performance impact with respect to specific metrics, the severity of overhead, etc.

### 5.6.9 Fuzzy Approach to Bottleneck Search

There are several tools supporting bottleneck search, e.g. in [55, 87]. These tools, however, support crisp-based searching as the search conducts based on checking crisp threshold. Given a performance metric, a threshold is predefined. During the search, the performance metric is evaluated against the threshold, and when the performance metric exceeds the threshold, a bottleneck is assumed to exist in the code region. The base metric used in the search can be performance measurements or performance properties [87]. There are two drawbacks of current crisp search strategy. Firstly, the search does not give the degree of severity of the bottleneck, e.g. *extremely bottleneck* or *slightly bottleneck* (performance property represents the severity of the performance, not of the bottleneck). Secondly, there is no support to specify in-exact bottleneck search statements such as *negligible bottleneck*. These statements are important as bottleneck threshold, by itself, is not exact value.

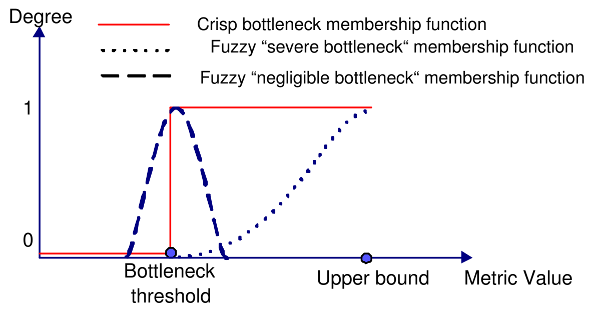
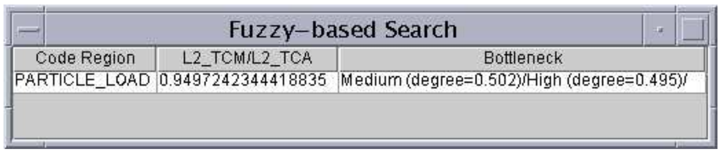


**Fig. 5.17.** Fuzzy versus crisp bottleneck search.

We propose fuzzy-based bottleneck search that addresses the drawbacks mentioned above. Figure 5.17 outlines the fuzzy-based bottleneck search. Given a threshold, we can use fuzzy sets to represent the severity of bottleneck and the negligible bottleneck range besides the fuzzy set representing the bottleneck threshold. For example, in Figure 5.17 we define a Pi-function fuzzy set used to check the negligible (close to) bottleneck points and S-function fuzzy set used to check the severity of bottleneck. When searching the bottleneck points, the value of metric used in bottleneck search is evaluated against these fuzzy sets. Not only we can locate which code region is a bottleneck point as usual but also we can provide the severity of bottleneck and determine negligible bottleneck points.

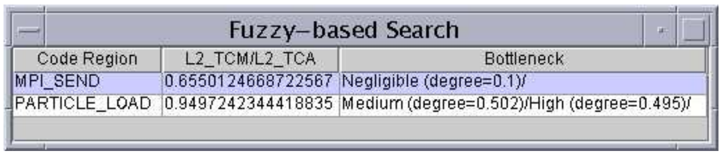
Very simply, to demonstrate the advantage of fuzzy-based bottleneck search, we experience with 3DPIC code to find code regions that may have L2 cache access problems. Suppose a code region whose L2 cache miss ratio exceeds 0.7 is a bottleneck. In the first case we use a set of performance characteristic terms $T = \{low, medium, high\}$ representing the severity of the bottleneck. Three different fuzzy sets Z-function with range $[0.7, 0.8]$, Pi-function with range $[0.75, 0.95]$ and S-function with range $[0.9, 1]$ are associated with *low, medium, high* term, respectively. We apply this search with 3DPIC code executed with 4 processes and we find that there is only one bottleneck as shown in Figure 5.18(a). The bottleneck falls into both classes *medium* and *high*, as shown in Figure 5.18(a). Without using fuzzy-based, we do not have information about the severity of the bottleneck. Since we are not certain about the threshold, we decided to use another triangle fuzzy set with parameter $(0.65, 0.7, 0.75)$ to describe close area of the pre-defined bottleneck threshold. Consequently, we find another code region as presented in Figure 5.18(b).



Fuzzy–based Search

| Code Region | L2_TCM/L2_TCA | Bottleneck |
|---|---|---|
| PARTICLE_LOAD | 0.9497242344418835 | Medium (degree=0.502)/High (degree=0.495)/ |

(a) Without negligible bottleneck search



Fuzzy–based Search

| Code Region | L2_TCM/L2_TCA | Bottleneck |
|---|---|---|
| MPI_SEND | 0.6550124668722567 | Negligible (degree=0.1)/ |
| PARTICLE_LOAD | 0.9497242344418835 | Medium (degree=0.502)/High (degree=0.495)/ |

(b) With negligible bottleneck search

**Fig. 5.18.** Results of bottleneck search based on fuzzy sets.

Given fuzzy sets and performance thresholds, tools can employ several searching methods to find the performance problems such as tree-based and tabu search. Currently, the main focus of our fuzzy-based bottleneck search is to describe and represent bottleneck conditions by using fuzzy sets, not to develop search engine. We are planning to adapt AKSUM search engine [87] to support our fuzzy-based bottleneck search.

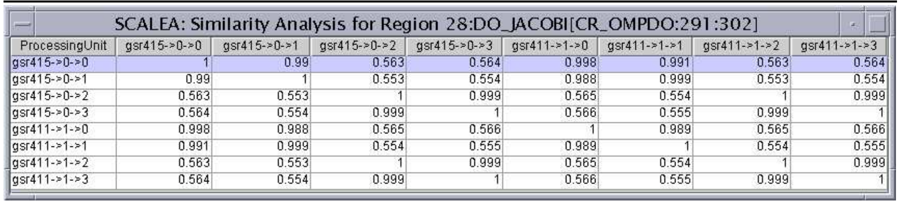
### 5.6.10 Similarity Analysis

When processing large and diverse performance data, information about performance similarities and differences in that data become more important because examining each data item in details is a time-consuming effort. However, existing performance tools lack support of similarity analysis.

#### 5.6.10.1 Similarity Analysis for Code Regions

We use similarity measure to compare performance similarity of code regions for single and multiple experiment(s). We implement two analysis tasks. Firstly, given a selected code region in one experiment, similarity measure is analyzed for all region summaries of the code region in processing units. Secondly, given a set of selected code regions and a set of experiments, we evaluate the similarity of selected code regions across experiments. The similarity measure is described in Section 5.6.5.1.

Figure 5.19 shows an example of similarity analysis for code region `DO_JACOBI` of Stommel with 2 processes (see Section 7.3.3 for more information

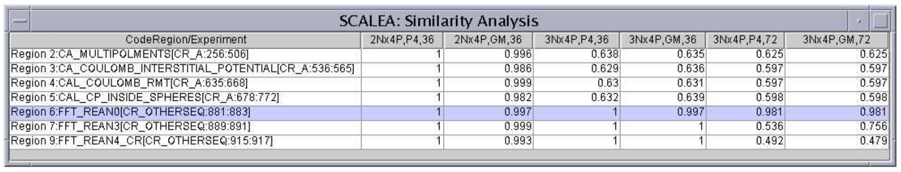


SCALEA: Similarity Analysis for Region 28:DO_JACOBI[CR_OMPDO:291:302]

| ProcessingUnit | gsr415->0->0 | gsr415->0->1 | gsr415->0->2 | gsr415->0->3 | gsr411->1->0 | gsr411->1->1 | gsr411->1->2 | gsr411->1->3 |
|---|---|---|---|---|---|---|---|---|
| gsr415->0->0 | 1 | 0.99 | 0.563 | 0.564 | 0.998 | 0.991 | 0.563 | 0.564 |
| gsr415->0->1 | 0.99 | 1 | 0.553 | 0.554 | 0.988 | 0.999 | 0.553 | 0.554 |
| gsr415->0->2 | 0.563 | 0.553 | 1 | 0.999 | 0.565 | 0.554 | 1 | 0.999 |
| gsr415->0->3 | 0.564 | 0.554 | 0.999 | 1 | 0.566 | 0.555 | 0.999 | 1 |
| gsr411->1->0 | 0.998 | 0.988 | 0.565 | 0.566 | 1 | 0.989 | 0.565 | 0.566 |
| gsr411->1->1 | 0.991 | 0.999 | 0.554 | 0.555 | 0.989 | 1 | 0.554 | 0.555 |
| gsr411->1->2 | 0.563 | 0.553 | 1 | 0.999 | 0.565 | 0.554 | 1 | 0.999 |
| gsr411->1->3 | 0.564 | 0.554 | 0.999 | 1 | 0.566 | 0.555 | 0.999 | 1 |

**Fig. 5.19.** Cache accesses similarity analysis for DO_JACOBI of Stommel executed on two 4-CPU SMP nodes. Similarity is measured with $\{$`(L2_TCA,1)`$\}$, membership function is `S-function`, and distance measure is based on Euclidean function. `gsr411->1-0` means thread 0 in process 1 in computational node `gsr411`.

about Stommel), each has 4 threads. This figure shows that the executions of `DO_JACOBI` in thread 2 and 3 of two processes are very similar. The executions in thread 0 and 1 are similar in process 0, but not in process 1. In all cases, the executions in thread 0 and 1 is quite different from that in thread 2 and 3. It suggests that there is a highly load imbalance between executions of `DO_JACOBI` in the same process. The load imbalance analysis confirmed this hypothesis.



SCALEA: Similarity Analysis

| CodeRegion/Experiment | 2Nx4P,P4,36 | 2Nx4P,GM,36 | 3Nx4P,P4,36 | 3Nx4P,GM,36 | 3Nx4P,P4,72 | 3Nx4P,GM,72 |
|---|---|---|---|---|---|---|
| Region 2:CA_MULTIPOLMENTS[CR_A:256:506] | 1 | 0.996 | 0.638 | 0.635 | 0.625 | 0.625 |
| Region 3:CA_COULOMB_INTERSTITIAL_POTENTIAL[CR_A:536:565] | 1 | 0.986 | 0.629 | 0.636 | 0.597 | 0.597 |
| Region 4:CAL_COULOMB_RMT[CR_A:635:668] | 1 | 0.999 | 0.63 | 0.631 | 0.597 | 0.597 |
| Region 5:CAL_CP_INSIDE_SPHERES[CR_A:678:772] | 1 | 0.982 | 0.632 | 0.639 | 0.598 | 0.598 |
| Region 6:FFT_REAN0[CR_OTHERSEQ:881:883] | 1 | 0.997 | 1 | 0.997 | 0.981 | 0.981 |
| Region 7:FFT_REAN3[CR_OTHERSEQ:889:891] | 1 | 0.999 | 1 | 1 | 0.536 | 0.756 |
| Region 9:FFT_REAN4_CR[CR_OTHERSEQ:915:917] | 1 | 0.993 | 1 | 1 | 0.492 | 0.479 |

**Fig. 5.20.** Similarity analysis for LAWP0 with 6 experiments and 7 code regions. We used `(wtime, 1.0)` to compute similarity measure, and distance measure is based on Euclidean function. Experiment `2Nx4P,P4,36` is selected as the base. `1Nx4P` means 1 SMP node with 4 processors. `P4` and `GM` correspond to MPICH CH_P4 and Myrinet, respectively. The problem size is either 36 or 72 atoms.

Figure 5.20 presents an example of using similarity analysis to examine selected code regions in 6 experiments. The first observation is that the performance of code region `FFT_REAN0` in the last 5 experiments is almost complete similar to the first experiment. The performance of `FFT_REAN3, FFT_REAN4` is almost similar in the first 4 experiments. This suggests that the performance of these code regions is not affected by changes of number of processors, communication libraries, even problem sizes (in case of `FFT_REAN0`). All code regions have similar performance in the first two experiments, suggesting the use of Myrinet does not help to increase much performance. This is confirmed by many cases in which communication libraries are very dissimilar but the performance is very similar.

| **Factor** | **Fuzzy Set** | **Range** | **Factor Category** |
|---|---|---|---|
| atoms | linear | [0,72] | problem size |
| CPU | S-function | [0,64] | machine |
| network | S-function | [0,158.20] | communication |

**Table 5.9.** Parameters for measuring similarity between controllable factors. In all test cases, maximum atoms is 72, maximum number of CPUs is 64, maximum network bandwidth benchmarked is 158.20 Mbytes/s.

| **Experiments** | **2Nx4P, P4,36** | **2Nx4P, GM,36** | **3Nx4P, P4,36** | **3Nx4P, GM,36** | **3Nx4P, P4,72** | **3Nx4P, GM,72** |
|---|---|---|---|---|---|---|
| $sim_{f_{atoms}}$ ({atoms,1}) | 1 | 1 | 1 | 1 | 0.5 | 0.5 |
| $sim_{f_{CPU}}$ ({(CPU,1)}) | 1 | 1 | 0.9531 | 0.9531 | 0.9531 | 0.9531 |
| $sim_{f_{network}}$ ({(network,1)}) | 1 | 0.1519 | 1 | 0.1519 | 1 | 0.1519 |
| $sim_o$ ({(wtime,1)}) | 1 | 0.996 | 0.638 | 0.635 | 0.625 | 0.625 |

**Table 5.10.** Example of similarity analysis with experiment factors for code region `CA_MULTIPOLMENTS` in 6 experiments. The similarity is measured with the first experiment as the base. The performance score of the code region is based on S-function. Distance measure is based on Euclidean function.

#### 5.6.10.2 Similarity Analysis for Experiment Factors

Our first step is to examine the similarity among various experiment factors and the performance of code regions. Table 5.9 shows an example of parameters of controllable factors including problem size, machine size, network. Table 5.10 presents the result of an example in which similarity is measured for for code region `CA_MULTIPOLMENTS` in 6 experiments of LAPW0 (see Section 7.3.5 for information about LAPW0) by using parameters in Table 5.9. In some cases, communication factor has very little impact on the performance. For example, the network between the first and the second experiment is quite dissimilar and other factors are very similar, but the performance of the code region is very similar. A similar result if we examine the fifth and sixth experiments. The CPU factor has significant impact on some cases. For example, factors of the third experiment are the same as those of the first experiment, except the CPU factor which is slightly different. However, the performance of the code region in the two experiments is quite different.

#### 5.6.10.3 Representing Similarity Measure in Words

Similarity measure, which is in the range $[0, 1]$, indicates the similarity degree. We can represent similarity measure by using fuzzy terms such as *complete dissimilar, very dissimilar, dissimilar, medium, similar, very similar* and *complete similar*. Each fuzzy term is associated with a fuzzy set describing a class of the similarity measure. For example, Table 5.11 presents the similar measures between the last 5 experiments with the first one in Table 5.10 by

| **Experiments** | **2Nx4P, GM,36** | **3Nx4P, P4,36** | **3Nx4P, GM,36** | **3Nx4P, P4,72** | **3Nx4P, GM,72** |
|---|---|---|---|---|---|
| $sim_{f_{atoms}}$ | complete similar | complete similar | complete similar | medium | medium |
| $sim_{f_{CPU}}$ | complete similar | very similar | very similar | very similar | very similar |
| $sim_{f_{network}}$ | dissimilar | complete similar | dissimilar | complete similar | very dissimilar |
| $sim_o$ | very similar | medium | medium | medium | medium |

**Table 5.11.** Example of using linguistic terms representing similarity measures.

using linguistic terms. The linguistic terms with their associated fuzzy sets are shown in Figure 5.21.

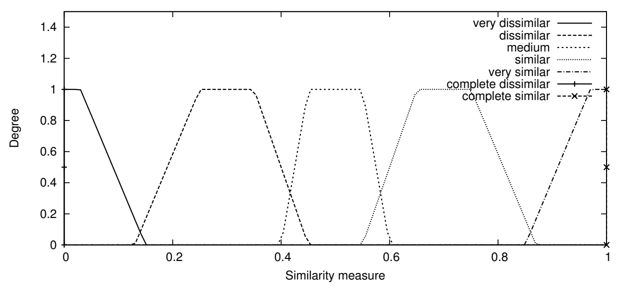


**Fig. 5.21.** Fuzzy sets describing similarity measure.

By using linguistic terms to describe the similarity measure, the performance analysis tool provides a high level abstraction of similarity measure to the end user, presenting a more friendly user interface with resulting similarity measure described in linguistic expressions.

### 5.6.11 Fuzzy Cluster Analysis

Cluster analysis [136, 2] is a core task in data mining [121] that is used to build or discover groups (clusters) in a data set. Group members are similar objects that share certain common properties thus the resulting classification may provide insightful information for discovering structures in the data clustered [136, 2]. Clustering is unsupervised process that does not use prior knowledge in the classification.

Recently, cluster techniques have been used to determine the performance characteristics, such as in [22]. Code regions are clustered into groups, and then tools or users can focus on analyzing and tuning code regions in interesting groups. However, current cluster techniques used in these tools have some limitations. Firstly, the classification of code regions normally is based on a single metric or a set of performance metrics without normalizing the metrics. Secondly, the clusters, e.g. produced by the k-means procedure, are crisp clusters, since any code region either is or is not a member of a particular cluster.

To overcome these limitations and to complement existing work on exploiting cluster techniques for performance analysis of distributed and parallel programs, firstly we employ the concept of performance score to represent the performance of code regions whose performance metrics may have different scales. Secondly, we introduce fuzzy C-means clustering that uses performance scores as metrics to build groups of code regions with similar performance and to determine the degree of membership of code regions in each cluster.

#### 5.6.11.1 Fuzzy C-Means Clustering

Given an experiment $e$ with region summary set $RS = \{rs_1, rs_2, \cdots, rs_n\}$, $rs_j \in \vec{R^p}$, $p$ is the number of performance metrics collected for $rs_j$, we can compute a set of performance score vectors $S$ of $n$ data points $\vec{s_j} \in \vec{R^p}$. Each $\vec{s_j}$ represents performance scores of a $rs_j$. The clustering algorithm aims to group $RS$ into $c$ clusters.

The detailed fuzzy C-means clustering algorithm is introduced in [42, 128]. In our implementation of the fuzzy C-means clustering, the clustering result is obtained by minimizing the following objective function

$$J(S,U,C) = \sum_{i=1}^{c}\sum_{j=1}^{n} u_{ij}^{m} d^2(c_i, \vec{s_j}) \tag{5.21}$$

subject to

$\forall i \in \{1, \cdots, c\} : \sum_{j=1}^{n} u_{ij} > 0$, and
$\forall j \in \{1, \cdots, n\} : \sum_{i=1}^{c} u_{ij} = 1$

where $u_{ij} \in [0,1]$ is the membership degree of the performance score vector $\vec{s_j}$ to the $i-th$ cluster, $c_i$ is the cluster center of the $i-th$ cluster, and $d(c_i, \vec{s_j})$ is the distance between $\vec{s_j}$ and $c_i$. $C$ denotes the set of all $c$ cluster centers $C = \{c_1, \cdots, c_c\}$. The matrix $Uc \times n = [u_{ij}]$ is called fuzzy partition matrix and the parameter $m$, $m > 1$, determines the degree of fuzziness. The distance function is

$$d^2(c_i, \vec{s_j}) = (\vec{s_j} - \vec{c_i})^T(\vec{s_j} - \vec{c_i}) \tag{5.22}$$

The algorithm repeatedly calculates fuzzy cluster centers and updates $U$. The calculation and update will be terminated when a partition matrix $\delta$ satisfies the condition $\delta \leq \epsilon$ where

$$\delta = \parallel U^{l+1} - U^{l} \parallel = max_{ij}|u_{ij}^{l+1} - u_{ij}^{l}| \tag{5.23}$$

and $\epsilon$ is a pre-defined threshold and $l$ indicates the computation step.

### 5.6.12 The Use of Fuzzy Rules for Data Reduction

When analyzing a large performance datasets, e.g. classifying or clustering code regions, we need to conduct data reduction because many computations are required in order to process a large set of performance measurements of multiple metrics.

Current performance tools do not focus on filtering the data to be processed. In many cases, we may detect code regions with poor performance but these code regions do not have a significant impact on the performance of the whole program. For instance, an MPI code region would be classified as poor communication because it has low transfer rate, but it may not be important if it takes only 0.0001% of the wall-clock time of the program. Thus, if we take into account how important the code region is, we can reduce the size of data to be processed, making analysis process faster and more scalable. Another example is the use rules, which are established based on previous performance data, to control the instrumentation of the next experiment. However, crisp conditions like those implemented in [166] do not deal with the uncertainty (of data and control).

We develop a method for reducing data collected that uses fuzzy-based rules. To utilize fuzzy rules to reduce data, we define performance characteristic terms representing filter conditions.

**Definition 5.3 (Data filter proposition).** *Let $s$ be the performance score of region summary $rs$ with respect to metric $m$, $\tau$ be a performance characteristic term representing the data filter condition. Region summary $rs$ should be excluded with respect to metric $m$ if fuzzy proposition ($s$ is $\tau$) yields true.*

**Definition 5.4 (Fuzzy-based rule for data reduction).** *Let $\vec{s}$ be performance scores of $rs$, $\{\tau_1, \tau_2, \cdots, \tau_p\}$ be performance characteristic terms for metrics $\{m_1, m_2, \cdots, m_p\}$, $\tau_i$ is the filter term for $m_i$. A fuzzy-based rule used to filter data can be defined based on fuzzy propositions built from $s_i$, $\tau_i$ as follows*

**IF** ($s_1$ is $\tau_1$) AND ($s_2$ is $\tau_2$) ... AND ... **THEN** (consequent)

where *consequent* is to exclude the region summary.

For example, we can define a rule as

```
IF (ncalls is very small) AND (wtime is very low) THEN filter
```

to exclude any code regions whose the number of calls (denoted by `ncalls`) is very small, and the wall-clock time (denoted by `wtime`) is very low from the clustering process.

The fuzzy rules can be used by any performance tasks that need to reduce irrelevant data from a large performance datasets. For example, we can use the rules to reduce data from trace files, data for performance classification, or to disable the instrumentation of code regions in one experiment based on performance data obtained in previous experiments.

## 5.7 Summary

In this chapter, we have presented a novel classification of temporal overhead which can be used to explain the performance problems occurring in parallel programs. We have discussed how overhead analysis for code regions is conducted. The use of the dynamic code region call graph (DRG) enables detailed overhead analysis for any code regions. We present a flexible instrumentation system which allows the user to customize the code regions and performance metrics of interest that should be measured. The overhead classification is the key tool which helps the instrumentation system to determine code regions which should be instrumented in order to measure performance overheads of interest.

We have described a novel design of SCALEA's experiment data repository which holds all relevant experiment information including source code, experiment description, performance information, system-specific data, etc., and presented capabilities to support search and filter of performance data, and multi-experiment performance analysis, and to facilitate the data sharing and tool integration. However, employing the data repository introduces extra overheads in comparison with other non-employing-data-repository tools; the overheads occur in filtering and storing raw data to and retrieving data from the database. In the current implementation, we observed the bottleneck when accessing the data repository with a large volume of data. We are going to enhance our access methods and database structure to solve this problem.

We present the soft performance analysis. Soft performance analysis techniques proposed exploit and combine the fuzzy logic and similarity theory, and machine learning algorithms to provide soft, scalable and intelligent techniques for analyzing and comparing the performance of large and complex parallel and distributed applications. We discuss several flexible and convenient methods to deal with uncertainty in performance analysis, e.g. fuzzy-based bottleneck search, and means for conducting performance analysis in the form closer to human notions, e.g. fuzzy-based search query.

# 6

# Performance Monitoring and Analysis for the Grid

## 6.1 Introduction

This chapter describes our methods and techniques for performance monitoring, instrumentation, measurement and analysis in the Grid. First, we present the design and implementation of a unified monitoring and performance analysis system for the Grid. We introduce a novel sensor-based middleware for performance monitoring and data integration in the Grid that is capable of self-management. The middleware unifies both system and application monitoring in a single system, stores various types of monitoring and performance data in decentralized storages, and provides a uniform interface to access that data. We have developed event-driven and demand-driven sensors to support rule-based monitoring and data integration. Grid service-based operations and TCP-based data delivery are exploited to balance tradeoffs between interoperability, flexibility and performance. Peer-to-peer features have been integrated into the middleware, enabling self-managing capabilities and supporting group-based and automatic data discovery, query and subscription of performance and monitoring data. The middleware is implemented as a set of Grid services based on the Open Grid Services Architecture (OGSA), providing an infrastructure for conducting online monitoring and performance analysis of a variety of Grid services including computational and network resources and Grid applications.

Second, we present a Grid service to support the dynamic instrumentation of Grid applications. The Grid dynamic instrumentation service provides a widely accessible interface to other services/users to control the instrumentation of Grid applications. The instrumentation service leverages an XML-based Standardized Intermediate Representation for Binary Code (SIRBC) for describing the program structure of executables, and an Instrumentation Request Language (IRL) for specifying code regions and performance metrics to be measured and for controlling the instrumentation task.

Third, we introduce a Grid service for online monitoring and performance analysis of scientific workflows in the Grid. The service collects resources monitoring data from Grid infrastructure monitoring, workflow execution status from the workflow control and invocation services, and performance measurements obtained through the dynamic instrumentation service. It then conducts the online analysis of these data along with the workflow graph. Relevant data to workflows including workflow graphs and performance data are stored. Novel techniques are developed to support multi-workflow analysis. Refinement constructs of workflows can be specified, and performance of refinement constructs of different workflows can be compared and evaluated for multiple experiments.

Finally, we propose a new approach to performance analysis, data sharing and tool integration in Grids that is based on ontology. We devise a novel ontology for describing the semantics of monitoring and performance data that can be used by performance monitoring and measurement tools. We introduce an architecture for an ontology-based model for performance analysis, data sharing and tool integration. At the core of this architecture is a Grid service which offers facilities for other services to archive and access ontology models along with collected performance data, and to conduct searches and to perform reasoning on that data.

This chapter is based on the work the presented in [253, 251, 248, 250, 249, 252].

## 6.2 A Unified System for Monitoring and Performance Performance Analysis in the Grid

Most existing Grid monitoring tools are separated into two distinct domains based on what they are monitoring: *Grid infrastructure monitoring* and *Grid application monitoring*. The lack of combination of two domains in a single system has hindered the user from correlating performance metrics of various sources at different levels when conducting the monitoring and performance analysis. Grid monitoring tools that are able to combine application and system monitoring and performance analysis are crucial as these tools will provide the user a unified view and support the correlation between performance metrics from various sources.

Most existing Grid monitoring tools focus on the monitoring and analysis of Grid infrastructure; yet little effort has been concentrated on performance analysis for Grid applications, especially for Grid scientific workflows. To date, application performance analysis tools are mostly targeted to conventional parallel and distributed systems (e.g. clusters and SMP machines). As a result, these performance analysis tools do not well address challenges in the Grid environment such as scalability, diversity, dynamics and security. Performance measurement, instrumentation and analysis for Grid applications require different approaches from those for conventional parallel and distributed systems. Moreover, Grid monitoring not only supports the end-user to monitor the Grid but also provides useful information for several functions such as performance analysis and tuning, performance prediction, fault detection and scheduling. Consequently, in order to support the interoperability among different Grid services, the performance monitoring and analysis service should be based on Grid service standards, e.g. OGSA.

### 6.2.1 Self-Managing Sensor-based Middleware

To monitor various resources in Grids, a large number of monitoring sensors needs to be developed and deployed in different domains. In our view, such sensors are very similar to those in sensor networks [23, 257] in which the sensor follows resource constraints such as communication (networks connecting sensors usually vary, having latency with high variance and sensors have to use limited bandwidth), computation (sensors have to use limited computing power and memory sizes, otherwise the monitoring may change the state of the monitored resources). These constraints limit data processing capability of a sensor thus normally the sensor sends collected data to a sink node which stores the data. In many cases the sink node also controls or requests data from sensors; we call such sink node a *sensor manager*. In Grids, it is impossible for all sensors to communicate with a central sensor manager. Also because resources on which sensors execute and resources sensors monitor may join and leave, the structure of sensor networks frequently changes. Therefore, sensors and sensor managers must operate in self-managed and decentralized manner.

Most existing Grid monitoring tools have monitoring sensors operating in distributed manner and the network connecting sensors to sensor managers exploits the various types of communication such as shared memory [32], TCP [269], UDP [36] and multicast [168]. However, these tools do not focus on the interoperability among sensor networks and the self-organization within them, and only support limited types of sensors. Mostly they support event-driven sensors (e.g. in [32, 168, 269]). Sensor managers are configured into tree of point-to-point connections (e.g. in [32, 168]); or directory services, supporting discovery of data and sensor managers, do not interact with each other (e.g. in [269]). Thus they do not cope well with Grid networks topology which changes frequently.

Lack of interoperability among sensor networks and lack of self-organization within them have hindered distributed data discovery, data query and subscription (DQS) in Grid monitoring tools, not to mention fault-tolerance. Currently data discovery and DQS are mostly based on hierarchical or centralized models, as surveyed in [104]. But such models do not work well with more dynamic and large-scale distributed environments in which useful information services are not known in advance. As it has been suggested, e.g. in [98, 231], and demonstrated, e.g. in [129, 232, 133], the super-peer model and service group, which are powered by peer-to-peer (P2P) computing [173], have the advantages in solving the above-mentioned issues, but have not been exploited in Grid monitoring middleware. Moreover, most Grid monitoring tools are not capable of self-configuration and -reconfiguration under varying conditions which occur frequently in the Grid. Autonomic computing [146] which aims to deal with the unpredictable conditions of systems should be exploited.

Integrating performance and monitoring data in Grids is crucial because it is likely that no single tool will be deployed to provide performance data for all Grid sites, while there is a need of utilizing and analyzing monitoring data across multiple Grid sites at the same time. Each Grid site is equipped with different computing capabilities, platforms, libraries that require various performance monitoring and measurement tools. However, Grid users should not be forced, when possible, to access monitoring data in Grids by using different mechanisms. Instead, the users should be able to utilize that diverse monitoring data by using the same mechanism.

Seamless integration and high interoperability require well-defined interfaces, rich expressive customized data representations, and more power to process and store data. However, the involvement of more functions and processing results in slower performance. Therefore, we need to balance tradeoffs between interoperability and performance. On the one hand, using Grid/Web service-based operations and XML data supports high interoperability among different tools, easily customizing collected data. However, the performance considerably suffers when data is delivered via Web service operations with SOAP [79]. On the other hand, (parallel) TCP-based data streams can be utilized to achieve higher performance in delivering data in Grids [24]. Current Grid monitoring tools exploit either purely Grid service-based operations or TCP-based data streams.

### 6.2.2 Sensor-based Middleware Overview

Figure 4.2 depicts the architecture of sensor-based middleware implemented in SCALEA-G with the main Grid services named Directory Service, Sensor Manager Service, Client Service. These services, based on OGSA [93] and organized into service groups, manage, store and provide various types of performance and monitoring data measured and gathered by an extensive set of distributed sensors. They are capable of self-management and they can collaborate in serving the requests from clients.

**Directory Service (DS)** stores information (e.g. schema and availability) about performance and monitoring data, Sensor Manager Service and other services of the middleware. **Sensor Manager Service (SM)** manages sensors and data collected and gathered by sensors, and provides these data to consumers via DQS operations. An SM can interact with several sensor instances executed on distributed machines; sensor instances will send their collected data to SMs. SM uses XML containers to store performance and monitoring data. **Client Service (CS)** provides interfaces for administrating activities of SMs, querying data registered in DS, subscribing and/or querying data stored in SMs, etc. Other services (e.g. scheduling service) access performance data by exploiting facilities provided by CS.

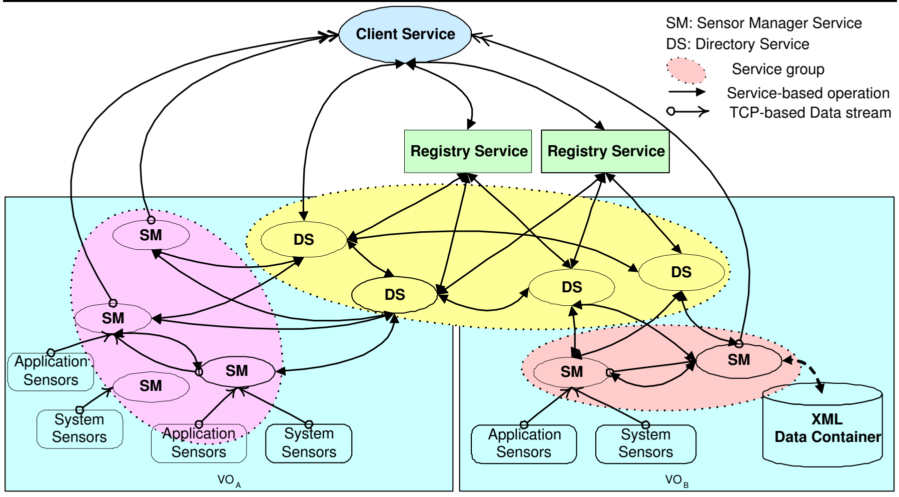


**Fig. 6.1.** High-level view of self-managing sensor-based middleware

SM and DS are organized into two types of groups (communities): *SM Group* and *DS group*. Within a Virtual Organization (VO) [95] there could be several SM groups. A DS group is deployed for multiple VOs; each VO provides a number of DSs which form the DS group. DSs register their information with a set of Registry Services. By using CS, the client of the monitoring middleware, which explores the monitoring service through existing Registry Services, can find DSs and SMs and then access performance and monitoring data. In our framework, we use existing implementations of Registry Service. However, DS and SM are specially designed for the performance and monitoring purpose.

### 6.2.3 Sensor Model for Performance Monitoring and Data Integration

In what follows, we present the concept of various types of sensors, execution models of sensors, Sensor Manager Service and interactions between sensors and Sensor Manager Services.

#### 6.2.3.1 Conceptualization of Monitoring Sensors

Figure 6.2 presents the underlying concept of our monitoring sensors. Sensors are used to capture performance data and to monitor *monitored resources* including computational and network resources, and Grid applications[1]. Every sensor monitors one or more *resources* and provides *monitoring data* (e.g.,

[1] a *monitored resource* is any object that can be monitored.

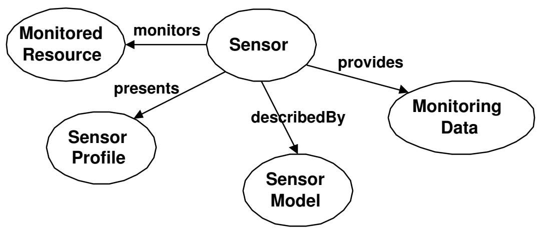


**Fig. 6.2.** Conceptualization of monitoring sensors.

events and performance measurements) of the monitored resources; each resource is determined by a unique resource identifier (`ResourceID`) and monitoring data is described in XML (each type of monitoring data is determined by an XML schema.). Each sensor presents a *sensor profile* which describes the sensor, e.g. unique sensor identifier (`SensorID`), sensor description and lifetime, how to control the sensor (e.g. calling parameters) and information about the sensor. How a sensor works is described by the *sensor model*, e.g. event-driven or demand-driven, or rule-based monitoring. A unique tuple (`SensorID`, `ResourceID`) is used to determine a type of monitoring data of a monitored resource.

A sensor can be invoked at different times with different parameters. Each invocation of a sensor is called a *sensor instance* referring to a particular instantiation of a sensor at run-time. Sensor instance delivers its collected data described in XML format to designated SMs. Each sensor instance has its own lifetime which is the period of time that the sensor instance is running. Different instances of a sensor can be used to monitor different objects and they might require different parameters. For example, a sensor which is used to measure the available bandwidth of a network path[2] in the Grid can have two instances: one used to monitor the bandwidth between host $A$ and $B$ and the other for host $C$ and $D$.

### 6.2.3.2 Event-driven and Demand-driven sensors

Sensors in most existing Grid monitoring tools are based on event-driven model: a sensor measures and collects data based on events, mostly time-based event. Event-driven sensors collect the data and store the collected data when an event happens at a time, without consideration at a time when an event happens the data is needed. Demand-driven sensors collect and provide data only when receiving requests. Demand-driven sensors are particularly useful for integrating data provided by other sources. To realize the importance of both types of sensors, our middleware supports both event-driven and demand-driven sensors.

[2] Network path is the network-level abstraction of a "virtual link"from/between host A to/and host B [192].

#### 6.2.3.3 System Sensors and Application Sensors

In our framework, we distinguish two types of sensors: *system sensors* and *application sensors*. System sensors are used to monitor and measure the performance of Grid computational resources and networks. Application sensors embedded in applications are used to measure execution behaviour of code regions and to monitor events in these services. This distinction allows us to simplify the management of two different types of sensors. For example, the security model applied to the access of data provided by system sensors is looser than that provided by application sensors.

Our system sensors are used to monitor computational resources and networks, and provide a set of predefined metrics of monitored objects. A system sensor may not really perform the measurement and monitoring; it can just query or collect data from others that perform the real measurement. Most system sensors provide dynamic information, for example the available bandwidth of a link, CPU time usage of a host, but some can provide static information, for example information of computational node. Our framework provides system sensors for monitoring the most commonly needed types of performance information on the Grid investigated by GGF DAMED-WG [77].

For instance, Figure 6.3 presents an excerpt of XML schema of the data provided by a sensor namely `path.bandwidth.capacity.TCP`. This sensor is used to measure the bandwidth capacity between two hosts in the Grid.

```
<xsd:complexType name="SensorData">
 <xsd:sequence>
  <xsd:element name="source" type="xsd:string"/>
  <xsd:element name="destination" type="xsd:string"/>
  <xsd:element name="eventtime" type="xsd:long"/>
  <xsd:element name="bandwidth" type="xsd:double"/>
 </xsd:sequence>
 <xsd:attribute name="SensorID" type="xsd:NMTOKEN"
                fixed="path.bandwidth.capacity.TCP"/>
 <xsd:attribute name="ResourceID" type="xsd:string"/>
</xsd:complexType>
```

**Fig. 6.3.** XML schema of data provided by path.bandwidth.capacity.TCP sensor.

Application sensors are embedded in user's programs via source code instrumentation prior to application execution or in application processes through dynamic instrumentation at the application runtime. Application sensors provide *profiling* and *user-defined event* data. Profiling sensors collect user-selective performance metrics (e.g. timing and hardware counters) of code regions in applications whereas user-defined event sensors generate specific events at selected points in programs when given conditions occur.

Application sensor instances are automatically invoked each time the application control flow reaches the relevant sensor code.

```
<xsd:complexType name="SensorData">
  <xsd:sequence>
  <xsd:element name="exp" type="xsd:string"/>
  <xsd:element name="pu" type="PU" minOccurs="0" maxOccurs="1"/>
   <xsd:element name="cr" type="CR" minOccurs="0" maxOccurs="1"/>
   <xsd:element name="metrics" type="MetricList"
                                  minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="SensorID" type="xsd:NMTOKEN" fixed="app.prof"/>
  <xsd:attribute name="ResourceID" type="xsd:string"/>
</xsd:complexType>
```

**Fig. 6.4.** Top-level XML schema of application profiling data.

Similar to system sensors, application sensor data is represented in XML format. For example, Figure 6.4 shows an excerpt of XML schema of application profiling data. The `SensorID` element specifies the name of the sensor, The `exp` element is a unique identifier determining the experiment. The identifier is used to distinguish data between different experiments. The `pu` element specifies the information about the processing unit on which the code region is executed. The `cr` element refers to information of the source of the code region (e.g. line and column). A profiling data message contains a list of performance metrics, each of which is represented in a tuple of (`name=value`).

#### 6.2.3.4 Sensor Repository

Normally, system sensors are shared by various users. To help the user to simplify the management and deployment of system sensors in SCALEA-G, we provide a *system sensor repository* and a framework for developing and deploying sensors in the repository. A developer writes a system sensor based on the system sensor template, describes properties of the sensor and specifies parameters needed to invoke the sensor. All of these information is then stored into the sensor repository managed by SMs which make sensors in the repository available for use when requested. A sensor repository then can be attached to a sensor hosting environment, for example a sensor manager that can make use of the sensors in the repository. By using reflection mechanism, the sensor manager can activate a sensor on-demand. The user makes requests *create sensor instance* by invoking a service operation of SMs or directly invokes system sensor to create a new instance of a system sensor. The lifetime of an instance can be specified when the sensor instance is activated.

Figure 6.5 presents an excerpt of XML schema used to express sensors in the sensor repository. We can specify static information of sensor profile such as `SensorID`, implementation class of the sensor (by the `measureclass` element), XML schema of data provided by the sensor (`schemafile` element), parameters needed to invoke the sensor (`params` element), etc. With enough information about the sensor described in the repository, a sensor can be easily activated by using Java reflection mechanism.

```
<xsd:element name="sensor" type="SensorDescription" minOccurs="0"
                                          maxOccurs="unbounded"/>
<xsd:complexType name="SensorDescription">
 <xsd:sequence>
    <xsd:element name="measureclass" type="xsd:string"/>
    <xsd:element name="desc" type="xsd:string"
                                      minOccurs="0" maxOccurs="1"/>
    <xsd:element name="schemafile" type="xsd:string"/>
    <xsd:element name="params"  type="ParamsEntry"
                                      minOccurs="0" maxOccurs="1"/>
 </xsd:sequence>
 <xsd:attribute name="SensorID" type="xsd:NMTOKEN"/>
 <xsd:attribute name="ondemand" type="xsd:boolean"/>
</xsd:complexType>
<xsd:complexType name="ParamsEntry">
 <xsd:sequence>
   <xsd:element name="param"  type="ParamEntry"  minOccurs="0"
                                             maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ParamEntry">
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="desc" type="xsd:string"/>
    <xsd:attribute name="dataType" type="xsd:string"/>
</xsd:complexType>
```

**Fig. 6.5.** Excerpt of XML schema used to describe sensors in the sensor repository.

#### 6.2.3.5 Sensor Manager Service

The main tasks of Sensor Manager Service (SM) are to control and manage activities of sensors in the sensor repository, to publish information about data collected by sensors to DSs, to receive and buffer monitoring data sensors produce, and to provide monitoring data to consumers via data query and subscription (DQS). SM includes the following components: Service Administration, Data Query and Subscription, Data Receiving and Publishing, as shown in Figure 6.6. SM provides control and DQS tasks, data receiving

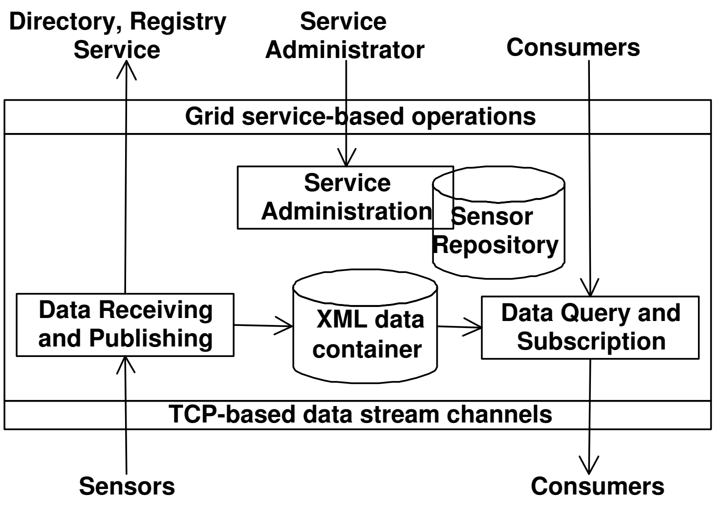

Directory, Registry Service
Service Administrator
Consumers
Grid service-based operations
Service Administration
Sensor Repository
Data Receiving and Publishing
XML data container
Data Query and Subscription
TCP-based data stream channels
Sensors
Consumers

**Fig. 6.6.** Sensor Manager Service Implementation

and delivery over both Grid-based service operations and TCP-based data streams.

The *Service Administration* component receives requests from the SCALEA-G administrator[3] and controls activities of SMs and their sensors in the sensor repository. The administrator can activate any sensor (thus making a new sensor instance) or deactivate an existing sensor instance. When a new sensor instance is activated, it will be added into the list of sensor instances. Conversely, a sensor instance will be removed from this list when it is deactivated. The administrator can perform the registration of (adding, updating or removing) information about the SM, properties of data provided by sensor instances with selected DSs. All the interactions in this component are carried out through invocations of Grid-based service operations.

The *Data Query and Subscription* component is responsible for processing DQS requests from consumers. DQS tasks are implemented as service operations but monitoring data is delivered via TCP-based data streams. Based on the information published in the DS, consumers can subscribe monitoring data provided by sensor instances that is archived in SMs. The resulting data will be *pushed* to consumers via TCP-based data channels. When a consumer unsubscribes a data type, the DQS component will stop sending that data to the consumer. To support the *pull* mode, this component processes queries with constraints (produced by consumers) to filter data of interest. The resulting data satisfying the requested constraints will be sent back to

[3] Hence, *administrator* refers to *SCALEA-G administrator* who administers the SCALEA-G services.

the consumer. DQS requests can be represented in XPath based on the published XML schema.

The *Data Receiving and Publishing* component conducts two tasks. First, it receives data collected by sensor instances. A data receiver is used to receive data from sensors. It then processes and stores the received data into data containers. The data receiver is a thread binding a well-known port and interacts with sensors via TCP connections. Data is stored in XML data containers. Second, it implements functionality that publishes (adding, updating, removing) information about SM and properties of monitoring data to DS.

#### 6.2.3.6 Interaction between Sensor and Sensor Manager Service

In order to allow sensors to freely customize their provided data, SM must receive, process and store multiple data types with unknown (XML) structures into its data containers. To do so, we design a generic three-phase protocol for exchanging data between sensors and SMs.

In this protocol, the interactions between sensors and SMs involve the exchange of three XML messages: `sensorinit` message in the initialization phase, `sensordataentry` message in the measurement phase, and `sensorfinal` message in the final phase. Measurement data is encapsulated into `sensordata entry` message. The measurement data is enclosed by `<![CDATA[ ... ]]>` tag. Thus, sensors can customize the structure of their collected data. Before it stops sending collected data, the sensor instance sends a `sensorfinal` XML message to notify the SM. The three XML messages that always contain `SensorID` and `ResourceID` with other information (e.g., XML schema of data which sensor produces, lifetime and description information about the sensor) are self-explained. Based on (`SensorID, ResourceID`), SM can setup appropriate buffers and store data into the buffers. Therefore, multiple types of monitoring data can be delivered via a transient connection, not just through a persistent one.

Sensors can send XML schemas to SM even though SM does not need the schemas in order to process the received data. SM will publish these schemas to DS so that other services consuming collected data can get schemas in order to make use of the data. Instead of sending XML schema to SM, the schemas can be stored at SM. Our approach allows any sensor that implements the above-mentioned protocol to send the data to SM while SM is not necessarily aware of the structure of the measurement data.

#### 6.2.3.7 Rule-based Monitoring

Different from event-driven sensors in existing Grid monitoring tools, our event-driven sensors can support rule-based monitoring. Instead of sending monitoring data it collects, the sensor uses rules to analyze monitoring data and reacts with appropriate functions.

Because different resources have different characteristics, we need to setup different rule sets for monitoring different resources. For example, with two different instances of a sensor used to monitor the bandwidth of two network paths which have different characteristics, e.g. one with maximum 100 MBytes/s, the other with 10 MBytes/s, the rules used to detect whether the bandwidth is low or high must be different even though the way to monitor these paths is the same. Therefore, a rule set is normally associated with each sensor instance. However, multiple sensor instances can share the same rule set, e.g. when their monitored resources have the same characteristics.

We use ABLE Rule Language (ARL) [60], which supports if-then-else rules, when-do pattern match rules, etc., to define rule sets for sensors. ABLE toolkit [44] provides a wide range of inference engines to process the ARL rule sets, e.g. boolean forward/backward chaining, fuzzy forward chaining and pattern match engine. For example, to define a fuzzy variable for monitoring bandwidth of a network path in the Austrian Grid [28], we used Iperf [134] to test the bandwidth, and obtained the maximum observed bandwidth which never exceeds 5 MBytes/s. We divided the bandwidth into 5 states by using fuzzy logic, as shown in Figure 6.7. Based on this fuzzy variable, we define a rule set, presented in Figure 6.8. With this rule set, depending on the status of bandwidth of the network path, e.g. *very low, low* or *very high*, the sensor will react with appropriate functions, e.g. sending events to SM.

```
Fuzzy bandwidth= new Fuzzy( 0,5) {
      Shoulder VERYLOW = new Shoulder(0, 1, ARL.Left);
      Triangle LOW = new Triangle(1, 1.5, 2);
      Trapezoid MEDIUM = new Trapezoid(2,2.2,2.8 ,3);
      Triangle HIGH = new Triangle(3, 3.5, 4);
      Shoulder VERYHIGH = new Shoulder(4, 5, ARL.Right);
    };
```

**Fig. 6.7.** A fuzzy variable describing status of the bandwidth of a network path.

In the case where rules are not specified when a sensor is instantiated, the sensor instance will work as in the normal model (e.g. sending monitoring data when the event happens). Rule-based monitoring approach has many advantages as it allows us to easily control and customize the monitoring actions. In addition, we can implement autonomic features that consider changing systems as an effect of the monitoring behavior. However, there is no common rule set for all resources even those monitored by a single sensor. Rules have to be built for each resources based on best practices.

```
S1: bandwidth = getBandwidth();
R_VERYLOW: if (bandwidth is VERYLOW) {
             doReactionWhenBandwidthVeryLow();
           }
R_LOW:     if (bandwidth is LOW) {
             doReactionWhenBandwidthLow();
           }
R_VERYHIGH: if (bandwidth is VERYHIGH) {
             doReactionWhenBandwidthVeryHigh();
            }
R_OTHER:    doNormalReaction();
```

**Fig. 6.8.** Example of rule set for bandwidth of a network path.

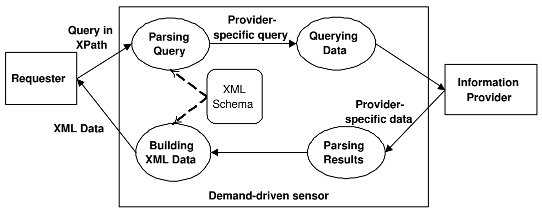


**Fig. 6.9.** Using a demand-driven sensor to integrate performance data.

#### 6.2.3.8 Performance and Monitoring Data Integration by Using On-demand Sensor

Besides using demand-driven sensors to monitor resources, we also exploit them for data integration. Figure 6.9 presents the model of using demand-driven sensor for integrating performance and monitoring data from other providers, e.g. MDS (Monitoring and Discovery System) [70], NWS (Network Weather Service) [269] and Ganglia [168]. To access different providers, we develop different demand-driven sensors taking the role of data mediators. As shown in Figure 6.9, when the sensor receives an XPath-based request from a requester, based on XML schema, it parses the request, extracting information of the request such as tag names with their associated attributes. The sensor then constructs a provider-specific request, calling the information provider with that request, and obtaining the result in provider-specific format. The sensor then parses this result and builds a new result described in XML. The XML-based result will be sent back to the requester. With this approach, other services use the same mechanism to access data in other providers as in our service.

When a demand-driven sensor is activated, the sensor returns information about resources whose monitoring data it can collect to the SM which in turn publishes the information to DSs. With that information, consumers can create requests for monitoring data. For instance, consider the `path.predict.bandwidth.capacity.TCP` sensor which obtains predicted network bandwidth from NWS. The startup information for an instance of this sensor is configured as follows:

```
<startupsensor name="path.predict.bandwidth.capacity.TCP">
 <startupparam name="NWSNameServer"
                      value="olperer.dps.uibk.ac.at:8090"/>
</startupsensor>
```

When this sensor starts it queries the NWS NameServer and returns network paths whose predicted bandwidth values it can collect to the SM. A consumer specifies an XPath-based request as follows

```
/sensordata[@SensorID="path.predict.bandwidth.capacity.TCP"]
[source="blindis.dps.uibk.ac.at:8060"][destination="olperer.
dps.uibk.ac.at:8060"]
```

in order to obtain the predicted bandwidth of network path `(blindis.dps.uibk.ac.at:8060, olperer.dps.uibk.ac.at:8060)`. This request is then translated into an `nws_extract` command as follows

```
nws_extract -f,time,mae_forecast -N olperer.dps.uibk.ac.
at:8090  bandwidthTcp blindis.dps.uibk.ac.at:8060 olperer.
dps.uibk.ac.at:8060
```

which is used to obtain predicted bandwidth values from NWS. The output of `nws_extract` is then translated into XML that is sent back to the requester. The requester only knows the XML schema of requested data in order to specify the request. The rest, where the requested data locates and how to get the requested data, are done by the middleware. Our integration approach aims at providing a uniform, flexible interface for accessing data collected by lower-level, domain-specific information providers.

### 6.2.4 Self-Organizing Services

#### 6.2.4.1 Service Group

Each SM or DS group has a set of operations associated with the group. These operations address (i) how the requests for performance data are handled, and (ii) how the requested data are delivered. The real number of members of a group is dependent on the actual deployment that can dynamically change. In the whole system, many different groups could exist.

The group operations associated with SM group are group-based DQS. One member of the group can act as a mediator for other members. Given a DQS request, an SM can provide requested data even though its storage does not contain the requested data by collaborating with other SMs in the same SM group. For a DS group, the group-based operation supports the discovery of data providers. Given a request for finding the provider of a needed data type, DSs in a DS group can cooperate in determining the data provider.

#### 6.2.4.2 Data Dissemination and Maintenance

Instances of sensors are executed in monitored nodes and send data collected to SM which in turn stores the data into its data container. SM automatically publishes characteristics of received data to a set of DSs, not to a single DS. Each SM keeps its group name and a list of DSs to which it publishes data. The list of DSs can dynamically change over the time. Each SM keeps a list of Registries through which it can search for information about DSs. When an SM is created, the SM gets a maximum number $p$ of DSs it should register with and a pre-defined updated interval $t$ seconds. In the cycle of $t$, SM lookups Registries to get $n$ DSs. The SM then selects `min(`$n$`,`$p$`)` DSs from $n$ ones. A DS is chosen based on the following procedures. DS is selected firstly based on domain-name approximation, and then based on the latency of `ping` operation, and finally based on random selection. SM then disseminates information about data it stores to its selected DSs.

A DS can publish information about itself to multiple Registries. In the current implementation, one DS belongs to a DS group. A DS keeps a list of Registries with which it registers its information. A DS maintains a list of DSs in its groups; these DSs are its edge peers. Repeatedly with predefined $t_r$ seconds, the DS searches Registries for up-to-date information about its edge peers. DS also performs `ping` test to its edge peers to check whether its edge peers are alive. To make sure that it provides the updated information, the DS checks its data in the database periodically based on a pre-defined $t_d$ seconds. During the checking procedure, DS invokes `ping` operation of registered SMs. If a `ping` to an SM failed, DS assumes the SM to be out of service and then DS removes all information associated with that SM. In the cycle of $t_u$ seconds, DS publishes its information to Registries. Before DS finishes its execution, it unregisters its information from the Registries.

As commonly in service-oriented computing, the availability of Registries, DSs and SMs is the key issue to the fault-tolerance of the middleware. Instead of storing data into a centralized SM, collected data are stored over a set of distributed SMs, thus guaranteeing that a failure of one or many SMs does not bring the whole service down. Differing from existing monitoring tools whose SM mostly publishes data to a single DS, our SM publishes its information to multiple DSs. Thus, not only data is widely disseminated and highly available but also it guarantees that if a DS is failed to serve requests from clients, still other DSs can do.

#### 6.2.4.3 Discovery of Data Providers

Any client that wants to subscribe or query performance data of a resource has to locate a corresponding SM which provides the data. The discovery of data providers is based on requests containing tuples of (`SensorID`, `ResourceID`). A tuple (`SensorID`, `ResourceID`) is unique that determines monitoring data of a resource. Each DS provides a set of operations for other services to retrieve and search its registered data. Based on a tuple (SensorID, ResourceID), a client can call operations of a DS in order to discover data providers registered with that DS. (SensorID, ResourceID) can also be specified in the data content filters of DQS requests.

Data discovery can also be done automatically by CS thus a client does not need to interact with DSs. CS parses client requests to get detailed elements such as SensorID and ResourceID. A list of DSs will be obtained from given Registries. The request is then sent to DSs which in turn cooperate to locate SMs by using group-based operations. When a DS cannot locate the provider of the requested data, it forwards the request to all its edge peers, otherwise it just sends back the results. These edge peers conduct the search and return the result to the caller. The DS then sends back the result to the requester. A parameter is used to control the request forwarding policy.

### 6.2.5 Data Query and Subscription

#### 6.2.5.1 Query and Subscription Operations

SM provides a set of service operations for other services to subscribe and query data available in the SM. Below, we just outline main operations.

- *subscribeData(consumer_dr, SensorID, ResourceID, content_filter, ResultID, duration, relay)*: consumer_dr specifies the *Data Receiver* of the consumer; the Data Receiver indicates the endpoint receiving the resulting data. A consumer performs a subscription of monitoring data of a resource determined by ResourceID; the data is collected by a sensor determined by SensorID. Parameter content_filter specifies the content filter applied to the requested data. SensorID and ResourceID are optional as they can be specified in the content filter. Subscription time is specified by a duration. ResultID specifies the identifier of the resulting data. If the subscription operation is successful, a subscription_id will be returned to the caller whereas resulting data is delivered to consumer via TCP-based streams. relay specifies whether an SM should relay the subscription to other SMs when it cannot serve the request.
- *unsubscribe(subscription_id)*: this operation is used to terminate an existing subscription.
- *renew(subscription_id, duration)*: renew or extend an existing subscription (determined by subscription_id) to new duration.
- *queryData(consumer_dr, SensorID, ResourceID, content_filter, ResultID, relay)*: similar to *subscribeData* operation but for querying data.

Content filters are described in XPath that can be easily written based on XML schemas of data provided by sensors.

By using operations of SM, CS supports both *one-to-one* and *one-to-many* DQS requests. In one-to-one mode, a subscription or query request is used to obtain performance data provided by a single SM whereas in one-to-many mode a client subscribes or queries data from many SMs by using a single subscription or query request.

#### 6.2.5.2 Automatic Query and Subscription

Clients can also perform DQS automatically without knowing where the requested data is located. The content filter, specified in DQS requests that clients pass to CS, can contain characteristics of data such as `SensorID` and `ResourceID`. Given an XPath-based content filter, we can obtain XML tags, attributes (e.g. `SensorID` and `ResourceID`), etc., of the filter by using an XPath parser. For example, by processing the following content filter

```
/sensordata[@SensorID='host.mem.used'][ResourceID='bridge.
vcpc.univie.ac.at']/...
```

, we obtain (`host.mem.used`, `bridge.vcpc.univie.ac.at`) as the value of (`SensorID`, `ResourceID`).

When receiving a request from clients, CS processes the content filter and obtains `(SensorID, ResourceID)` information. It then searches DSs in order to find SMs that provide the requested data; the search is mentioned in Section 6.2.4.3. CS then sends requests to SMs which provide the requested data. As a result, the client does not necessary know where the monitoring data is stored. If DQS requests contain information about sensors and monitored resources, the middleware can automatically handle DQS requests.

The middleware provides simplified APIs for clients to conduct automatic querying and subscribing performance and monitoring data. The APIs hide all the lower-level details of the middleware. For example, Figure 6.10 presents a simple code which is used to query available monitoring data of CPU usage of the machine `schareck.dps.uibk.ac.at`. The `ConsumerService` class, a part of CS, is responsible for processing DQS tasks. The client knows a Registry Service. It indicates the service handle of Registry Service (variable `handle`), specifies the content filter (variable `content_filter`), and calls the CS. CS returns a `DataSensorReader` from which the resulting data is retrieved. The client can call APIs (blocking and non-blocking) or set up a call-back on `DataSensorReader` to obtain the data.

#### 6.2.5.3 Group-based Data Query and Subscription

An SM can act as a mediator for other services to access data provided by other SMs in its group. When a client sends a DQS request to an SM, if the SM does not provide the requested data, SM will search its registered DSs to find SMs that can serve the request. If the search is successful, the SM acts as a super-peer between the data requester and the SM provider by forwarding the request to the SM provider. The provider first tries to communicate with the requester. If successful, the provider sends requested data to the requester,

```
ConsumerService cs = null ;
cs = new ConsumerService();
cs.activateUpDataService();
String handle="http://bridge.vcpc.univie.ac.at:8765/ogsa/
                 services/samples/registry/VORegistryService";
String content_filter="/sensordata[@SensorID=\"host.cpu.used\"]
                     [@ResourceID=\"schareck.dps.uibk.ac.at\"]";
SensorDataReader out =
      cs.distributedQueryDataWithRegistry(handle, content_filter);
```

**Fig. 6.10.** Example of querying monitoring data by using information from Registry Service.

otherwise it sends data back to the caller SM. If an SM receives request from another SM, it will not propagate the request when it cannot serve the request.

In this model, an SM can take the role of the super-peer in either/both forwarding requests or/and delivering data. Any SM may become a super peer at runtime.

#### 6.2.5.4 Notifications

In all mentioned DQS, the client conducts DQS based on available information about monitoring data published in DS. However, there are many cases in which the client wishes to subscribe for a notification of interesting data which is not available at the time of the subscription. For example, the client may inform the monitoring system that it wishes to receive execution status of activities of a workflow application being executed by the workflow enactment before it submits the workflow application. We call this type of data subscription *notification subscription*.

DS and SM provide two service operations named `subscribeNotification`, `unsubscribeNotification` for subscribing and unsubscribing notification data. DS and SM use a table to keep existing subscriptions of notifications. The client can subscribe the notification on a specific SM or on the whole monitoring system. If the client wishes to receive notification message from a specific SM, the client can register with the SM by calling `subscribeNotification` operation of that SM. In this case, the client will not receive notification data collected by other SMs even though that data satisfies the client's request.

In our framework, SM gathers and stores performance data collected from sensors. However, there is no mechanism to determine SMs which are capable of distributing a specific notification data because SMs and sensors can enter and exit the monitoring system arbitrarily. The client may only be aware of a few services to which it contacts, e.g. a DS or an SM, but it wishes to receive a notification without knowing the service which is capable of providing this notification. We support this type of notification subscription by implementing a global notification mechanism. By using the CS, the client registers with a set of DSs $\{DS_1, DS_2, \cdots, DS_n\}$ that it knows, and indicates information about interesting data which it wishes to be notified. Each $DS_i$ updates the table containing subscriptions of notifications and then calls `subscribeNotification` operation of registered $\{SM_{i1}, SM_{i2}, \cdots, SM_{im}\}$ in its directory with that indicated information. Similarly, when a new SM registers with a DS, the DS calls that operation of the SM with existing subscriptions in its table. When receiving a `subscribeNotification` call, the SM updates a table containing tuples of `(ResultID, Subscription)`. Whenever SM receives data satisfying notification constraint, SM delivers the data to CS. If SM cannot deliver a notification to a client, the SM will remove the subscription of that notification from the table. To unsubscribe a notification, CS sends unsubscription requests to DSs which in turn pass these requests to SMs.

As each SM registers its information with multiple DSs, an SM can receive duplicate subscriptions for a notification. Currently, SM accepts the duplication because ensuring SM not to receive duplicate subscriptions of a notification might be at the higher cost than calling `subscribeNotification` which just updates the table containing subscriptions of notifications.

### 6.2.6 Communication, Data Delivery and Aggregation

#### 6.2.6.1 Service-based Operations and TCP-based Data Delivery

Each SM can be viewed as a peer in a P2P network. It, however, also is a Grid service. In most P2P systems, a peer processes requests and delivers data via TCP/UDP channels. Our peer is unique as we try to integrate both concepts, P2P model and Grid service, into a single peer. A peer provides Grid service operations for other peers and high-level clients to access and control its service. But, peers use TCP-based streams to deliver monitoring data to each other, thus relay functions can easily be implemented to support data delivery among peers. That also offers a higher performance for data delivery.

Figure 6.11 depicts how requests for data and requested data are handled. CS or SM requests data through *Grid service-based invocations* whereas requested data is delivered via *TCP-based streams. Data Sender, Data Receiver* and *Data Relay* of SM and CS are responsible for sending, receiving, and relaying performance data, respectively. Each Data Receiver or Data Relay is associated with an *endpoint* describing the network transport. An endpoint is described by a unique XML message containing network transport information such as host name, port, etc. An SM has only one connection to a consumer for delivering all kinds of subscribed data. The connection is created at the first subscription and will be freed after pre-defined $t_\delta$ seconds since the last subscription finishes. For delivering resulting data of queries, an on-demand connection will be created and freed when the delivery finishes. A data request always includes a unique `ResultID` which is associated with requested data that satisfies the subscribed/queried constraints. SM uses `ResultID` to route requested data to the destination while CS uses `ResultID` to aggregate results of the same request delivered from multiple SMs.

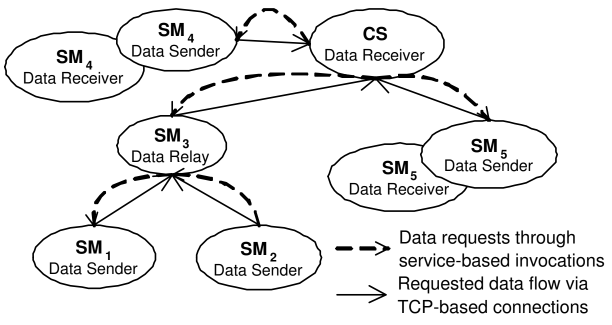


**Fig. 6.11.** Service-based invocations and TCP-based data streams.

In our middleware, monitoring data is described in XML. While using XML to describe performance and monitoring data provides a widely accessible interface and simplifies the interoperability among services, XML data grows in size. In subscription mode, the size of monitoring data delivered to the consumer each time is small and almost unchanged for a given subscription. However, in data query mode, the size of monitoring data can be large, depending on the query, for example, the consumer may want to retrieve resource monitoring data for the last 10 hours. To reduce the size of monitoring data transfered, we compress monitoring data before sending it over the network.

Data compression may increase the throughput and transfer rate. In Table 6.1, we monitored the data size and transfer time of CPU usage data from an SM in UIBK domain to a client in PAR domain (see Section 7.4.1 for more detail about the experimental test-bed); transfer time is the average value of observed values at different times. In our measurement, compression ratio, compression and decompression time are all dependent on the data size and type of monitoring data. When the data size is small, compressing data will not reduce transfer time because of the small compression ratio and the impact of compression and decompression time on the overall transfer time. However, when the data size is large, compressing data reduces the data size substantially that eventually improves both data transfer time and throughput significantly. For most types of monitoring data supported, when the size of data to be transfered is less than 512 bytes, the compression does not achieve a better transfer time and throughput because the compression ratio is close to 1. Moreover, there is an extra overhead due to the compression and decompression of data. Therefore, we develop a simple self-adaptive mechanism for deciding whether the resulting data should be compressed before sending to the requester that is based on the size of delivered data. If the data size is larger than $s_\delta$, the data will be compressed, otherwise data is transfered as normal. Currently, $s_\delta$ is set to 512 bytes.

| **Data size (bytes)** | $T_f$ **(ms)** | $T_{fcd}$ **(ms)** | $r$ | $T_c + T_d$ **(ms)** |
|---|---|---|---|---|
| 588 | 110 | 109 | 1.863 | 2 |
| 1560 | 119.33 | 115.24 | 4.537 | 2.11 |
| 3019 | 120.45 | 114.85 | 8.137 | 2.19 |
| 3991 | 120.68 | 114.68 | 10.207 | 2.44 |
| 5449 | 129.2 | 116.15 | 13.257 | 2.53 |
| 6421 | 130.45 | 116.05 | 14.967 | 2.73 |
| 7879 | 131.63 | 117.85 | 17.316 | 2.76 |
| 8851 | 136.28 | 117.48 | 18.712 | 3.08 |
| 10309 | 141.39 | 117.68 | 20.742 | 3.09 |
| 11281 | 143.25 | 117.93 | 21.949 | 3.22 |
| 12739 | 147.83 | 117.5 | 23.548 | 3.42 |
| 13711 | 149.75 | 117.25 | 24.355 | 3.67 |

**Table 6.1.** Example of transfer time without compression ($T_f$), transfer time of compressed data ($T_{fcd}$), compression ratio ($r$), compression and decompression time ($T_c + T_d$) for CPU usage data. Time is measured with Java `System.currentTimeMillis()` call. Compression and decompression are implemented based on `java.util.zip` package.

#### 6.2.6.2 Data Aggregation

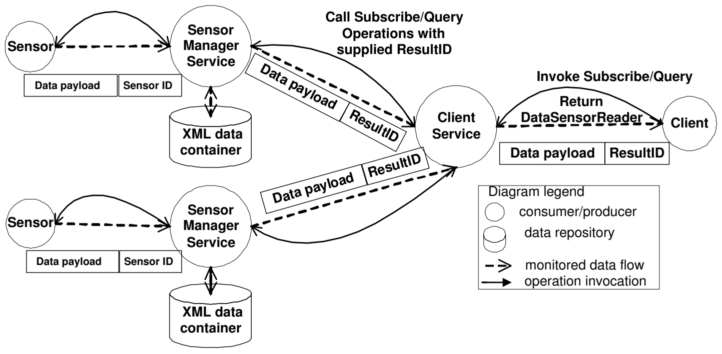


**Fig. 6.12.** Data aggregation model.

Data aggregation uses a simple mechanism as presented in Figure 6.12. In this protocol, each sensor constructs its XML data messages and sends the messages to an SM which stores the messages into appropriate buffers; each buffer uses a data container to store data. When a client subscribes and/or queries data by invoking operations of CS, CS calls corresponding operations of SM and passes a `ResultID` to the SM. CS returns client a `DataSensorReader`. The SM builds XML messages by tagging the `ResultID` to the data met the subscribed/queried condition and sends these messages to CS. At CS side, based on `ResultID`, the messages are filtered and stored into `DataSensorReader`. The `ResultID` is used to aggregate results of the same request delivered from multiple SMs. Clients can call operations (blocking and non-blocking) of or set up a call-back on `DataSensorReader` in order to get the resulting data.

#### 6.2.6.3 Buffering and Filtering Data

SM stores monitoring data in its native format; data is stored exactly as when it is sent to SMs. Data produced by system sensors will be cached in circular bounded buffers at SM. In the current implementation, for each type of system sensor, a separate data buffer is allocated for holding data produced by all instances of that type of sensor. In *push* model, any new data entry met the subscribed condition will always be sent to the subscribed consumers. In *pull* model, SM only searches current available entries in the data buffer and entries met conditions of consumer query will be returned to the requested consumers. Buffering data produced by application sensors is similar to that for system sensors. However, we assume that there is only one client to perform the monitoring and analysis for each application and the size of the data buffer is unbounded.

### 6.2.7 Security Issues

SCALEA-G requires authentication and authorization in several actions including registrations and queries to DSs, data queries and subscriptions to SMs, controls of activities of SMs and sensors. Each user has a public key based X.509 identify certificate. The authentication and authorization are carried out by using Grid Security Infrastructure (GSI) provided by Globus [100]. Data transfers between various components are performed via sockets based on GSI. The security employs both message-level and transport-level security.

#### 6.2.7.1 Access Control List

For some sensor instances, information collected has to be delivered to selected consumers. For example, the information collected by the application sensor instances has to be accessed only by the user who invokes the application. Therefore, SCALEA-G services must control over the user who is allowed to use the services. In the case SCALEA-G services are started by the administrator for the use of multiple users, each SCALEA-G service (e.g., DS and SM) contains an Access Control List (ACL) which maps user's information to properties of provided data the user can access. The ACL can statically be set up by the administrator. The user information obtained from user's certificate when the certificate is used in authentication will be compared with entries in the ACL in the authorization process.

#### 6.2.7.2 Security for Application Monitoring Data

In the case of application monitoring and performance analysis, when the user invokes the client GUI to subscribe/query data provided by application sensors, user's information will be recorded. Similarly, before application sensor instances start sending data to the SM, the SM obtains information of the user who executes the application (in initialization step). Both sources of information will be used for authorizing the user in receiving data from application sensors. We discuss this topic more details in Section 6.3.4.2.

## 6.3 Grid Service for Dynamic Instrumentation

While existing Grid toolkits provide core services for job submission, resource discovery, such similar Grid services for instrumenting Grid applications do not exist. In most cases, the instrumentation of Grid applications must be carried out by the end user. For applications that are executed on a single Grid site (within a single organization), existing instrumentation systems may be reused. However, for applications executed across multiple Grid sites, currently, the user has to manually instrument his code in order to obtain performance measurements of code regions of Grid applications because existing instrumentation systems are not appropriate. Consider the diversity and dynamics of the Grids. On the one hand, if the user wants to instrument his code, the user has to know in advance the Grids he submits jobs to, and has to select the right instrumentation tool for each Grid site. As a result, the user has to do a daunting task in order to instrument his code. Moreover, the selected instrumentation tool may not work with the monitoring middleware deployed in the selected Grid system. On the other hand, instrumentation techniques are typically bound to specific languages and systems. Therefore, it is possible that we need many different instrumentation systems just for instrumenting an application executed on Grids. More importantly, Grid workflows (WFs) tend to be composed from deployed components whose source code is not available. Without the instrumentation of code regions of workflow activities themselves, we are only able to monitor at the level of activity, thus significantly reducing the ability to detect and correlate performance problems.

We argue that the instrumentation service should be a core service of a Grid site. This approach gives many advantages. Firstly, an instrumentation service is bound to a specific Grid site, thus it can be better developed and can efficiently exploit features on that site. Since instrumentation services are autonomous, they are better to be coupled with the supportive monitoring middleware. Secondly, as an instrumentation system is a Grid service, the user does not need to worry about how to select an appropriate instrumentation system. Instead, he just discovers the service and uses it. Each Grid site may provide an instrumentation service that allows the user or the high level tools to control the instrumentation. The instrumentation service hides all the low-level details of the instrumentation process while the client of the instrumentation service just simply specifies its requests. To this end, the instrumentation service must support widely accessible interfaces, e.g. Grid/Web service operations, and protocols, e.g., SIR [219] and MIR [218] proposed by APART working group. Nevertheless, with such generic Grid instrumentation services, we have to accept some losses, e.g. instrumentation of arbitrary code regions.

### 6.3.1 Instrumentation and Measurement Techniques for the Grid

One of the key issues of the performance analysis of Grid applications is how performance data is measured and collected. Firstly, we have to study different instrumentation mechanisms to efficiently measure different types of performance data. Source code instrumentation provides a simple and efficient way for collecting measurement data, however, it requires the availability of all the source files. The instrumented sources have to be compiled and linked with instrumentation libraries for the specific target machines. That is a time consuming effort as each time the application executes the resources allocated may be different, not to mention the allocated resources may not be known in advance. Moreover, instrumentation and measurement metrics could not be changed during the runtime of the application. Dynamic instrumentation is complex but well-suited for measuring volatile and long-running applications, and for applications whose source code is not available. The workflow-based application (WFA) is normally dynamically composed from deployed applications whose source code is not available for instrumentation. The dynamic instrumentation would be an alternative for solving the problems arisen from the selection of instrumentation and measurement system and from the compilation of instrumented code fitted to the allocated resources.

We believe that instrumentation for the Grid should employ both methods. We can instrument sources of WF control and invocation service in order to gather execution status of WFs because execution status information is normally simple and small. However, for instrumentation of Grid applications, we believe that dynamic instrumentation would be more suitable. While source code instrumentation for Grid applications is widely supported, e.g., in [33, 118], little effort has been spent to investigate the dynamic instrumentation in the Grid, even though supporting dynamic instrumentation of parallel programs has a long history [172, 75].

Secondly, we have to carefully select the granularity of the measurement for Grid applications, namely profiling or tracing mechanism. Although many tools have support tracing of Grid applications, e.g. [197, 118], because Grid performance monitoring and analysis have to be conducted in online manner, tracing is not suitable for the Grid due to the fact that it generates a huge volume of trace data that has been transfered on the fly to the analysis component. On the other hand, traditional profiling is not suited for online monitoring and analysis because profiling data can only be obtained at the end of the execution of applications. Therefore, incremental mechanisms, for example profiling data is updated or requested and retrieved incrementally at runtime, would be more suitable.

### 6.3.2 Standardized Intermediate Representation for Executable Programs

#### 6.3.2.1 Standardized Intermediate Representation for Fortran, Java, C and C++ programs

Application performance analysis relies on performance information that is commonly obtained by executing instrumented applications; application can be statically instrumented before or dynamically instrumented during the execution. Normally the performance tool developers have to build separate instrumentation engines for different programming languages and for different instrumentation strategies such as dynamic and static instrumentation. This work is a tedious and time consuming effort. The APART working group has proposed a Standardized Intermediate Representation (SIR) as an abstract program representation for procedural and object-oriented programs [219]. Basically a SIR contains information about statement and directive types with very little details on the structure of individual statements and directives. The idea is that high-level tools would only know the types of a statement in order to make a decision about code regions that should be instrumented.

SIR is an XML-based representation that includes information about program units (e.g functions, methods), code regions (e.g. function calls, loops, statements), etc. SIR is designed for describing Fortran, Java, C and C++ in source code format. Therefore, it supports a rich set of information which is available when parsing a source code programs. However, with dynamic instrumentation, in which the intended instrumented program is available in binary code only, the information obtained is substantially reduced.

#### 6.3.2.2 SIR for Binary Code

At the level of binary code, mostly we obtain only information about program units, function calls and loops. Although we may measure information at the level of instructions, the information however is almost impossible to be mapped to the application. Therefore, by using that information we hardly reveal performance problems of the applications. As a result, the full SIR is not suitable as it requires the instrumentation engine to support many features that cannot be obtained with dynamic instrumentation. We therefore develop a simplified version of SIR for binary code (SIRBC) generated by C/C++ and Fortran compilers.

```
<xsd:element name="sirbc"   type="SIR"/>
<xsd:complexType name="SIR">
  <xsd:sequence>
    <xsd:element name="unit" type="SIRUnit"
                          minOccurs="0"  maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="SIRUnit">
  <xsd:sequence>
    <xsd:element name="coderegion" type="SIRCodeRegion"
                            minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="type" type="xsd:string"
                                    minOccurs="0"  maxOccurs="1"/>
  <xsd:attribute name="name" type="xsd:string"
                                     minOccurs="0"  maxOccurs="1"/>
  <xsd:attribute name="id" type="xsd:string"
                                     minOccurs="0"  maxOccurs="1"/>
</xsd:complexType>
<xsd:complexType name="SIRCodeRegion">
 <xsd:sequence>
  <xsd:element name="callee" type="SIRCallee"
                                      minOccurs="0"  maxOccurs="1"/>
 </xsd:sequence>
 <xsd:attribute name="type" type="xsd:string"/>
  <xsd:attribute name="id" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="SIRCallee">
  <xsd:sequence>
      <xsd:element name="linestart" type="xsd:integer"
                                      minOccurs="0"  maxOccurs="1"/>
      <xsd:element name="lineend" type="xsd:integer"
                                      minOccurs="0"  maxOccurs="1"/>
      <xsd:element name="sourcefile" type="xsd:string"
                                      minOccurs="0"  maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>
```

**Fig. 6.13.** Excerpt of XML schema of SIRBC.

Figure 6.13 presents the XML schema of SIRBC which is based on the idea of SIR. Currently SIRBC supports only level of program units, function calls. An application process is represented as a set of program units (element `SIRUnit`). Each program unit contains a set of code regions (element `SIRCodeRegion`). A code region is a function call that contains information (e.g. name, source information) about the calling function (element `SIRCallee`). Each program unit, code region is associated with a unique identifier(element `id`). The requester uses that identifier to indicate a program unit or a code region. By using SIRBC, the instrumentation requester can understand the application structure and create instrumentation requests specifying code regions that should be instrumented.

### 6.3.3 Instrumentation Request Language (IRL)

The IRL is provided in order to facilitate the interaction between instrumentation requesters (e.g. consumers, users, tools) and the instrumentation service. IRL is an XML-based language describing instrumentation requests and responses. Requesters send requests to instrumentation services and receive instrumentation responses describing status of the requests.

IRL requests allow specifying: (i) application process to be instrumented and the experiment identifier associated with performance measurements, and (ii) instrumentation requests. Figure 6.14 presents the XML schema of the current version IRL. The job to be instrumented is specified via `experiment` element. Information about the job includes application process identifier, application name, activity name (for workflow activity), and experiment identifier. The first three parameters are used to determine the application processes to be instrumented. Experiment identifier is used to associate performance measurements with the performance experiment.

The element `request` specifies types of IRL requests. Each request has a unique name. A request may consist of tasks; each task is associated with a set of performance metrics and code regions that will be affected by the task. Current IRL supports requests including *attach, getsir, instrument, deinstrument, finalize.*

- *attach*: requests the instrumentation service to attach an application and to prepare to perform other tasks on that application.
- *getsir*: requests the instrumentation service to return SIRBC of a given application.
- *instrument*: specifies code regions (based on SIR) and performance metrics should be instrumented and measured.
- *deinstrument*: specifies requests for deinstrumentation.
- *finalize*: notifies the instrumentation service that requester will not perform any request on the given application.

The code region `id` is used to determine the code region that should be instrumented.

```
<xsd:complexType name="IRL">
 <xsd:sequence>
  <xsd:element name="experiment" type="IRLExperiment" minOccurs="0" maxOccurs="1"/>
  <xsd:element name="request" type="IRLRequest" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="response" type="IRLResponse" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="IRLRequest">
  <xsd:sequence>
    <xsd:element name="experiment"  type="IRLExperiment" minOccurs="0" maxOccurs="1"/>
    <xsd:element name="task"   type="IRLTask" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NMTOKEN"/>
</xsd:complexType>
<xsd:complexType name="IRLExperiment">
 <xsd:sequence>
  <xsd:element name="applicationName" type="xsd:string"/>
  <xsd:element name="jobID"           type="xsd:string"/>
   <xsd:element name="activityID"      type="xsd:string"
minOccurs="0" maxOccurs="1" />
  <xsd:element name="experimentID"    type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="IRLTask">
 <xsd:sequence>
  <xsd:element name="coderegion" type="CodeRegion" minOccurs="1" maxOccurs="unbounded"/>
  <xsd:element name="metrics" type="MetricList" />
 </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="CodeRegion">
    <xsd:attribute name="unit"   type="xsd:string"/>
    <xsd:attribute name="name"   type="xsd:string"/>
    <xsd:attribute name="id"  type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="IRLResponse">
   <xsd:sequence>
      <xsd:element name="detail" type="xsd:string"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:NMTOKEN"/>
   <xsd:attribute name="status" type="xsd:NMTOKEN"/>
</xsd:complexType>
<xsd:simpleType name="MetricList">
  <xsd:list itemType="xsd:string"/>
</xsd:simpleType>
```

**Fig. 6.14.** Excerpt of XML schema of Instrumentation Request Language.

In responding to a request from a requester, the instrumentation service will reply to the requester by sending an instrumentation response which contains the name of the request, the status of the request (e.g OK, FAIL) and possibly a detailed responding information encoded in <`![CDATA[ ...
]]`> tag.

Figure 6.15 gives an example of IRL requests. The `experiment` specifies information about the process being instrumented. The request `instrument` demands the instrumentation service to conduct two instrumentation tasks. The first task specifies the instrumentation of code region `f1631_ionization`, which is a function call of `ionization` in unit `mymain`. The second task requests to instrument code region `f1439_pmpi_send`, a call of `pmpi_send` in unit

```
<?xml version="1.0" ?>
<irl>
<experiment>
   <applicationName>/home/truong/projects/rpp3d</applicationName>
   <jobID>14322</jobID>
   <experimentID>3DPIC-2N-4P</experimentID>
</experiment>
<request name="instrument">
 <task>
  <coderegion unit="mymain" name="ionization" id="f1631_ionization"/>
  <metrics>wtime</metrics>
 </task>
 <task>
   <coderegion unit="mymain" name="pmpi_send" id="f1439_pmpi_send"/>
   <coderegion unit="mymain" name="pmpi_recv"/>
   <metrics>wtime</metrics>
 </task>
</request>
<request name="finalize">
</request>
</irl>
```

**Fig. 6.15.** Example of IRL.

mymain, and to instrument all calls of pmpi_recv in unit mymain. All code regions are instrumented with wall-clock time (wtime). The finalize request indicates that the client will not conduct any instrumentation requests on the application process specified by the experiment element.

### 6.3.4 Grid Dynamic Instrumentation Service

Figure 6.16 presents the architecture of our dynamic instrumentation service for Grids. There are four main components residing in different places involving in the instrumentation process: *Instrumentation Requester* (IR), *Instrumentation Mediator* (IM), *Mutator Service* (MS) and *Instrumentation Forwarding Service* (IFS). The IR controls the instrumentation process. The MS, executed on the computational node where the application processes run, is responsible for performing the dynamic instrumentation. It attaches the application processes and inserts *application sensors* into the application processes. In the middle of IR and MS are the IM and the IFS which bridge and aggregate requests and responses between the IR and the MS. IM and IFS are needed because the IR cannot always directly communicate with the MS. IR works at a high-level at which it considers the execution of application as a whole. Therefore, IR may conduct instrumentation spanning multiple Grid sites. However, MS works at the lower level at which its monitored objects are application processes; an MS is executed on a computational node. Therefore,

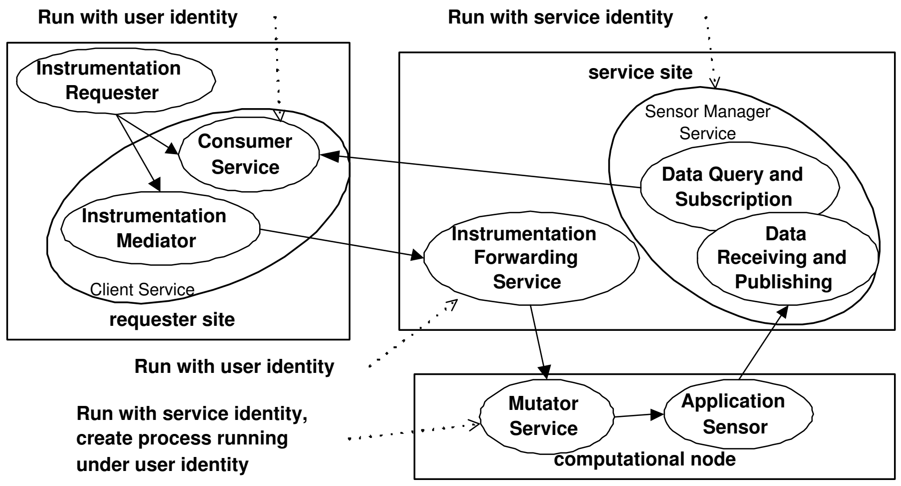

Run with user identity
Run with service identity
Instrumentation Requester
Consumer Service
Instrumentation Mediator
Client Service
requester site
service site
Sensor Manager Service
Data Query and Subscription
Data Receiving and Publishing
Instrumentation Forwarding Service
Run with user identity
Run with service identity, create process running under user identity
Mutator Service
Application Sensor
computational node

**Fig. 6.16.** Architecture of the Grid service of dynamic instrumentation

IM and IFS are used to transfer and aggregate requests and responses between the high-level view and the low-level one. An IFS instance is responsible for forwarding requests to multiple MSs executed on computational nodes. The above architecture is a service-oriented model built based on two languages: SIRBC and IRL. SIRBC allows the MS to describe instrumented applications in a neutral representation and to provide that representation to IR. IRL allows IR to define which portions of an application should be instrumented and which performance metrics should be collected.

The MS is a Grid service which is implemented based on gSOAP, a C++ Web Service toolkit with GSI-plugin [117]. MS accepts instrumentation requests represented in IRL. Figure 6.17 shows interactions between IR, MI, IFS, and MS instances when conducting requests for instrumenting an application. At the requester side, the IR specifies requests and passes these requests to IM. Based on the requests, the IM locates existing IFSs which can forward the requests to MSs executed on the same computational nodes of application processes; if no such IFSs exist, IM makes a request to create new IFS instances. IM then sends IRL requests to IFSs. When an IFS receives a request, it will search MS instances which can fulfill the request. If there is no MS instance for instrumenting application processes of a user in a computational node, IFS makes a request of creating a new MS instance for the user on that node. IFS will send the requests to MSs which in turn forward the requests to corresponding MSs. The MS will parse the IRL request and then perform the instrumentation of application processes. The MS inserts application sensors into application processes. The dynamic instrumentation techniques are facilitated by Dyninst [51]. The application sensors perform the monitoring and measurement of application processes. Performance measurement will be sent to Sensor Manager Service (SM), which is a part of the supportive monitoring middleware, or will be collected through MS.

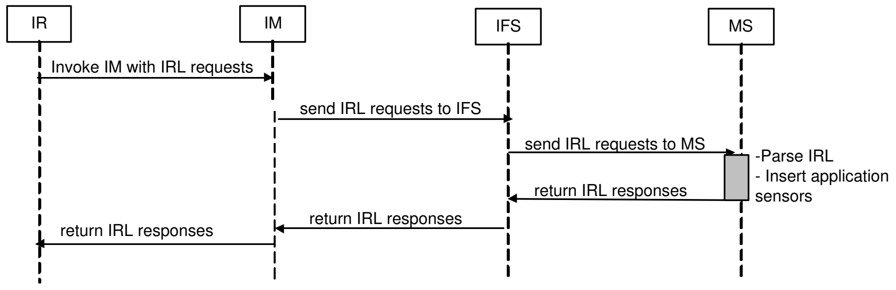


**Fig. 6.17.** Steps in conducting a request for instrumentation.

The MS provides the application structure to the requester in SIRBC format. Based on SIRBC, the IR can decide which code regions should be instrumented. With the high-level encapsulation and highly interoperability, interfaced through service operations, IRL language and SIRBC, the dynamic instrumentation service is widely accessible to other services.

### 6.3.4.1 Service Interface

The design of MS is based on the *factory model.* The MS consists of a Mutator Factory (MF) and Mutator Instance (MI). An MF is a persistent service deployed in each computational node. Figure 6.18 and Figure 6.19 present an excerpt of interfaces of MF and MI, respectively.

The MF provides a main operation named `createMutatorInstance` for creating MIs when requested. The MI is responsible for attaching application processes and instrumenting these processes. Information about MF is published to the supportive monitoring middleware. When IRF receives an instrumentation request, it finds MIs on corresponding computational nodes which can instrument application processes of the calling user. If no such an MI exists, the IFS calls the MF on the corresponding node to create a new MI. When an MI running, it connects to an SM, notifies its existence to the SM and waits for control from requesters. MI provides the following main operations:

- `performIRL`: to process IRL requests. The MI will react with appropriate functions such as attaching the application process, instrumenting and deinstrumenting, or detaching the application process.
- `getProfilingData`: to return profiling data collected to the requester.
- `destroyInstance`: to end the execution of this instance. When this operation is called the MI will free resources it occupies and finish its execution.

In addition, MF and MI provide two auxiliary operations: `ping` operation to support ping service and `getUserProcess` to obtain user processes executed on a computational node.

```
<wsdl:portType name="MutatorFactoryPortType">
  <wsdl:operation name="createMutatorInstance" parameterOrder="in0">
    <wsdl:input message="intf:createMutatorInstanceRequest"
                            name="createMutatorInstanceRequest"/>
     <wsdl:output message="intf:createMutatorInstanceResponse"
                            name="createMutatorInstanceResponse"/>
   </wsdl:operation>
   <wsdl:operation name="getUserProcess" parameterOrder="in0">
     <wsdl:input message="intf:getUserProcessRequest"
                                   name="getUserProcessRequest"/>
     <wsdl:output message="intf:getUserProcessResponse"
                                  name="getUserProcessResponse"/>
   </wsdl:operation>
   <wsdl:operation name="ping">
     <wsdl:input message="intf:pingRequest" name="pingRequest"/>
     <wsdl:output message="intf:pingResponse" name="pingResponse"/>
   </wsdl:operation>
 </wsdl:portType>
```

**Fig. 6.18.** Excerpt of interfaces of Mutator Factory.

#### 6.3.4.2 Security Model

The security in the dynamic instrumentation service is based on GSI [267] facilities provided by Globus Toolkit (GT) [108]. As shown in Figure 6.16, the security model employs both transport and message level security, using delegation, authentication/authorization, and run-as mechanism [4]. Except that MS uses transport level security, the interactions among the rest components are based on message level security. Message level security employs GSI secure conversation mechanism [4].

IR and IM run with the security identity of the user. IFS service methods are set to run with the security identity of the client. When IM requests IFS service to create an instance, the instance will be run with the security identity of the user. MF runs with the service identity in a none-privilege account. However, if MF is deployed to be used by multiple users, it must be able to create its instances running in the account of calling users. The MI created by MF upon on requests of IFS will be run as user identity. MF uses a grid-map file to authorize its requesters. As MI executes with the security identity of the user, it has permission to attach user application processes,

```
<wsdl:portType name="MutatorServicePortType">
   <wsdl:operation name="performIRL" parameterOrder="in0">
     <wsdl:input message="intf:performIRLRequest"
                              name="performIRLRequest"/>
     <wsdl:output message="intf:performIRLResponse"
                              name="performIRLResponse"/>
   </wsdl:operation>
   <wsdl:operation name="destroyInstance" parameterOrder="in0">
       <wsdl:input message="intf:destroyInstanceRequest"
                              name="destroyInstanceRequest"/>
       <wsdl:output message="intf:destroyInstanceResponse"
                              name="destroyInstanceResponse"/>
   </wsdl:operation>
   <wsdl:operation name="getUserProcess" parameterOrder="in0">
      <wsdl:input message="intf:getUserProcessRequest"
                                name="getUserProcessRequest"/>
      <wsdl:output message="intf:getUserProcessResponse"
                                name="getUserProcessResponse"/>
   </wsdl:operation>
   <wsdl:operation name="getProfilingData" parameterOrder="in0">
      <wsdl:input message="intf:getProfilingDataRequest"
                               name="getProfilingDataRequest"/>
      <wsdl:output message="intf:getProfilingDataResponse"
                              name="getProfilingDataResponse"/>
   </wsdl:operation>
   <wsdl:operation name="ping">
      <wsdl:input message="intf:pingRequest" name="pingRequest"/>
      <wsdl:output message="intf:pingResponse" name="pingResponse"/>
   </wsdl:operation>
</wsdl:portType>
```

**Fig. 6.19.** Excerpt of interfaces of Mutator Instance

and is able to perform the dynamic instrumentation. Delegation is performed from IM to IFS to MI.

In push mode, application sensors send measurements to SM. When subscribing and/or querying data provided by application sensors, data requester's identity will be recorded. Similarly, before application sensor instances start sending data to the SM, the SM obtains the security identity of the requester who executes the application. Both sources of information will be used for authorizing the requester which receives data from application sensors. In pull mode, performance measurements collected by applications sensors will be returned to the requester by MI. MI uses self-authorization mechanism to check the requester. Requests for obtaining performance measurements sent by IR will be delegated from IM to IFS to MI. As a result, only the owner can be able to access performance data.

#### 6.3.4.3 Practical Issues in Building SIR and Instrumenting Applications

When processing different binary codes compiled by different compilers, we observed that depending on specific compilers and architectures, the SIR of an executable is quite different from that of the other. It contains many internal functions that the user may not want to instrument. SIR, however, is designed for C/C++/Fortran/Java sources, thus, it does not define filters that can be used to exclude these irrelevant information when building the SIR from applications. We extend IRL to allow the IR to specify filters for `getsir` requests. The filters include code region names that the instrumentation service should exclude and the function scope in which the instrumentation service should limit its traversal.

Due to the dependence of executable structures on the compilers and platforms, the SIR of different processes of the same program may be different when the program is compiled and executed on different platforms. Thus, a SIR is associated with a process, not with a program. In some cases, the same code region has different identifiers in different SIRs. Thus when using identifiers to specify selected code regions, the IR has to process each SIR of a process individually. Consider a large number of processes, it is a time-consuming task for IR, if IR wants to instrument a code region in all processes. To avoid that, we can specify only the code region name and the program unit in instrumentation requests. The instrumentation service will instrument all functions which have that name within the given program unit.

### 6.3.5 Incrementally Updating Profiling Data

Traditionally, profiling is performed offline and performance measurements are summarized and available for being analyzed when the application finishes. Thus, this approach is not suitable for online profiling as we have complete summarized measurements only when the application finishes. Online profiling requires measurement data to be collected and analyzed during runtime of the application. But if summarized data is sent back to the analysis component at the instant the measurement data is updated, a huge data volume will be sent over the network. As a result, the impact of monitoring on the execution of the application is high.

We develop a mechanism to support online and incrementally updating profiling data. That is, instead of always updating consecutive measurements of code regions, the monitoring delivers measurement data to the analysis component incrementally. The monitoring system returns only the most-updated measurements in maximum pre-defined time or upon on a request. To profile a code region $r$ we put a sensor, composed by a start probe and a stop probe, as follow:

```
sis_start($PB_r$)
    $r$
sis_stop($PB_r$)
```

where $PB_r$ is information used to determine the code region;$PB_r$ is associated with a record storing measurement data of code region $r$. When an activation of $r$ finishes, its measurement data will be updated into the record. Each process keeps a profiling data of all instrumented code regions.

The analysis component can obtain the profiling data through *pull* or *push* mode. In pull mode, profiling data is stored in shared memory. The analysis component calls the `getProfilingData` operation of MI in order to obtain the requested profiling data.

```
procedure sis_start($PB_r$))
begin
    start the measurement of $r$.
    if (it is first execution of $r$) then
      send $PB_r$ to DRP component of SM.
    end if
end
procedure sis_stop($PB_r$))
begin
    stop the measurement of $r$.
    update performance measurements in $PB_r$.
    if ($PB_r$ is not in $buf_n$) then
      add $PB_r$ into $buf_n$.
    else
      update $PB_r$ in $buf_n$.
    end if
    if ($buf_n$ is full) then
      flush whole $buf_n$ to DRP.
      reset $buf_n$.
    end if
end
```

**Fig. 6.20.** Updating profiling data to Sensor Manager.

In push mode, the most recent updated measurements of $n$ code regions are stored into a *flush buffer* size $n$, $buf_n$. Performance measurements are incrementally sent to Data Receiving and Publishing (DRP) component of SM (see Figure 6.16 and Section 6.2.3.5). Figure 6.20 presents the algorithm used to send measurement data to the monitoring middleware. Every $t$ seconds since the last time the buffer is flushed to DRP, the buffer will be flushed if it is not empty. With this algorithm, performance measurements of $n$ last executed code regions are flushed to DRP incrementally in maximum $t$ seconds. As a result, we ensure that the requester receives the newly-updated profiling measurement of a code region no longer than $t$ seconds since the measurement is updated.

We have already implemented the push mode and are currently implementing the pull mode. In pull mode, application sensors are designed to store measurements in shared memory whereas those in push mode store measurements into internal buffers and push these measurements through the network. We are currently investigating to develop our application sensors so that they store collected data into shared memory. The task to support pushing or pulling profiling data will be done by MI. Also the `getProfilingData` operation will support requests based on MIR [218].

## 6.4 Performance Monitoring and Analysis of Grid Workflow-based Applications

Performance monitoring and analysis of Grid workflows (WFs) should support:

- **inter-activity performance monitoring and analysis**: to monitor and analyze the interactions between activities, the impact of an activity on the performance of the whole workflow or of the workflow construct that the activity participates. To this end, the monitoring and analysis tool has to operate at the level of the overall workflow and on the whole resources on which the workflow activities are executed.
- **intra-activity performance monitoring and analysis**: to monitor and analyze the performance of the invoked application of the individual activity. To this end, the monitoring and analysis tool has to operate at the level of the individual activity and on the resource on which the activity is executed.

Figure 6.21 presents the architecture of the Grid monitoring and performance analysis service for WFs. The WF is submitted to the *Workflow Invocation and Control* (WIC) [4] which locates resources and executes the WF. Events containing execution status of activities, such as *queuing* and *processing*, and information about resources on which the activities are executed will be sent to the monitoring tool. The *Event Processing* processes these events and the *Analysis Control* decides which activities should be instrumented, monitored and analyzed. Based on information about selected activity instances and their consumed resources, the Analysis Control requests the *Instrumentation and Monitoring Control* to perform the instrumentation and monitoring. Monitoring and measurement data obtained are then analyzed. Based on the result of the analysis, the Analysis Control can decide what to do in the next step.

[4] WIC is part of the Workflow Management System (WfMS).

This architecture uses SCALEA-G middleware as its supportive monitoring middleware. Various types of performance data are published to, stored in, and retrieved from SCALEA-G.

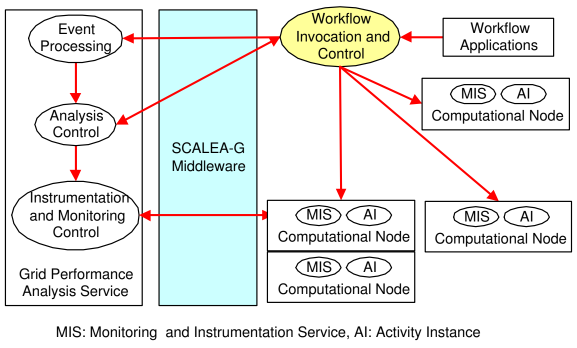

Event Processing
Analysis Control
Instrumentation and Monitoring Control
Grid Performance Analysis Service
SCALEA-G Middleware
Workflow Invocation and Control
Workflow Applications
MIS
AI
Computational Node
MIS: Monitoring and Instrumentation Service, AI: Activity Instance

**Fig. 6.21.** Model of monitoring and performance analysis of workflow-based application.

### 6.4.1 Supporting Workflow Computing Paradigm

Currently we focus on the workflow modeled as a DAG (Direct Acyclic Graph) because DAG is widely used in modeling scientific workflows. A WF is represented as a DAG of which a node represents an activity (task) and an edge between two nodes represents the dependency between the two activities. An activity instance may be executed on a single or multiple resources. Meanwhile we focus on activities whose invoked applications are application executables (e.g. MPI program).

We particularly concentrate on analyzing (i) *fork-join* model and (ii) *multi-workflow* of an application. Figure 6.22(b) presents the fork-join model of workflow activities in which an activity is followed by a parallel invocation of $n$ activities. This model is typical in many scientific WFs. There are several interesting metrics that can be obtained from this model, such as load imbalance, slowdown factor and synchronization delay at the synchronization point. These metrics help to uncover the impact of slower activities on the overall performance of the whole structure. We also focus on fork-join constructs that contain *structured block* of activities. A structured block is a single-entry-single-exit block of activities [5]. For example, Figure 6.22(c) presents structured blocks of activities.

An workflow-based application (WFA) can have different versions, each represented by a WF. For example, Figure 6.22 presents an application with

[5] More details of existing WF constructs can be found in [20].

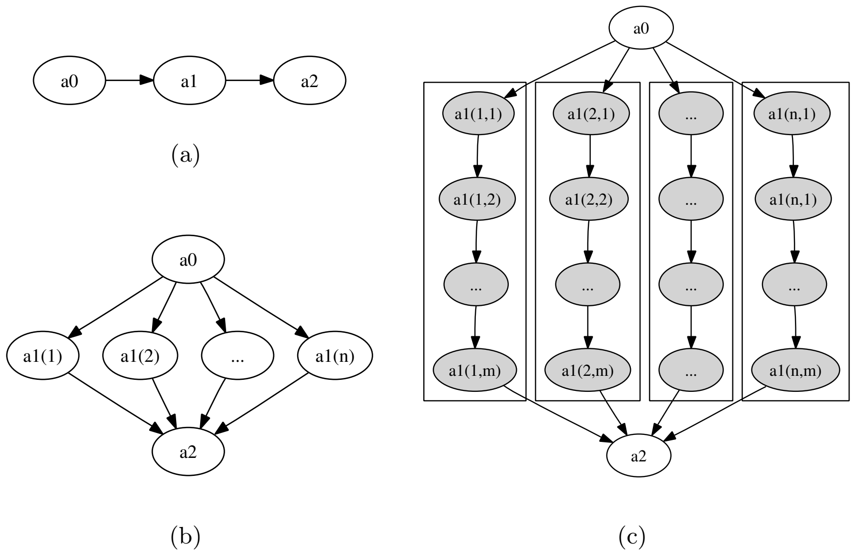


**Fig. 6.22.** Multiple workflows of an workflow-based application: (a) Sequence workflow, (b)Fork-join workflow, and (c) Fork-join structured block of activities.

3 different WFs, each may be selected for execution on specific underlying resources. When developing a WFA, we normally start with a graph describing the WF. The WFA is gradually developed in a sequence of *refinement* steps (stepwise refinement) that creates a better version or an adapted version fitted to a particular underlying Grid system[6]. This refinement can be done automatically by a workflow construction tool or manually by a WF developer. In a refinement step, a subgraph may be replaced by another subgraph of activities, resulting in a set of different constructs of the WF. For example, the activity $a1$ in Figure 6.22(a) is replaced by set of activities $\{a1(1), a1(2), \cdots, a1(n)\}$ in Figure 6.22(b). (Also we can consider a set of activities $\{a1(1), a1(2), \cdots, a1(n)\}$ is reduced to $a1$.) We call such refinement *refinement by replacement*. In Grids a WF can yield the best result in one particular run but not in the next run because the Grid infrastructure may be different from run after run. The concept of the best solution is now associated with a particular run. Moreover, since the underlying system changes from experiment to experiment a single WF may not be enough. As a result, different solutions for a WFA, even all of them are just used to conduct the same problem, may equally be important. The key question is which WF construct is the best for a given collection of resources. Therefore, multi-workflow analysis, the analysis and comparison of the performance of different WF con-

[6] In other words, a refinement step creates another solution for the WFA.

structs, ranging from the whole WF to a specific construct (e.g. a fork-join, a sequential construct), is an important feature.

We focus on the case in which a subgraph of a DAG is replaced by a another subgraph in the refined DAG. This pattern occurs frequently when developing WFs of an application for different underlying resource topologies. Let $G$ and $H$ be DAG of workflow $WF_g$ and $WF_h$, respectively, of a WFA. $G$ and $H$ represent different versions of the WFA. $H$ is said to be a *refinement* of $G$ if $H$ can be derived by replacing a subgraph $SG$ of $G$ by a subgraph $SH$ of $H$. The replacement can be controlled by the following constraints:

- Every edge $(a, b) \in G$, $a \notin SG$, $b \in SG$ is replaced by an edge $(a, c) \in H$, $\forall c \in SH$ satisfies no $d \in H$ such that $(d, c) \in SH$.
- Every edge $(b, a) \in G$, $a \notin SG$, $b \in SG$ is replaced by an edge $(c, a) \in H$, $\forall c \in SH$ satisfies no $d \in H$ such that $(c, d) \in SH$.

$SH$ is said to be a *replaced refinement graph* of $SG$. Note that $SG$ and $SH$ may not be a DAG nor a *connected graph.* For example, consider the cases of Figure 6.22(a) and Figure 6.22(b). Subgraph $SG = \{a1\}$ is replaced by subgraph $SH = \{a1(1), a1(2), \cdots, a1(n)\}$; both are not DAG, the first is a trivial graph and the latter is not a connected graph. Generally, we assume that there are $n$ components of a subgraph $SG$. Each component is either a DAG or a trivial graph. Comparing the performance of different constructs of a WFA can help to select and map WF constructs to the selected Grid resources in an optimal way.

Graph refinement is a well-established field and it is not our focus. Therefore, we do not concentrate on the determination of refinement graphs in workflows, rather, the workflow developers and/or workflow construction tools are assumed to do this task. Our main goal is that given different solutions for a WFA we study the performance similarity and difference between them.

### 6.4.2 Activities Execution Model

Each invoked application of an activity instance may be executed on different resources allocated by the WIC. Let $P(a)$ be an activity execution status graph modeling the execution of activity $a$ (hence we call the *execution graph* of an activity). A $P(a)$ is a directed, acyclic, bipartite graph $(S, E, A)$, in which $S$ is a set of nodes representing *activity states*, $E$ is a set of nodes representing *activity events*, and $A$ is a set of edges representing ordered pairs of activity state and event. Simply put, an agent (e.g. WIC, activity instance) causes an event (e.g. submit) that changes the activity state (e.g. from queuing to processing), which in turn influences the occurrence and outcome of the future events (e.g. active, failed). Figure 6.23 presents an example of an activity execution status graph. Note that the real execution model of a WF is more complex, depending on the implementation of WIC. For example, an activity can be re-submitted, aborted and suspended[7].

[7] Detailed possible states of a workflow can be found in [16].

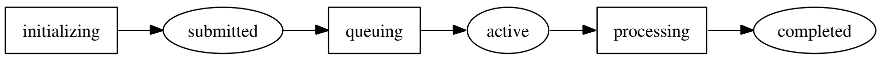


**Fig. 6.23.** Example of an activity execution status graph. □ represents a state, ◯ represents an event.

| Event Name | Description |
|---|---|
| active | indicate the activity instance has been started to process its work. |
| completed | indicate the execution of the activity instance has completed. |
| failed | indicate the execution of the activity instance has been stopped before its normal completion. |
| submitted | indicate the activity has been submitted to the scheduling system. |

**Table 6.2.** Example of event names.

Each state $s$ of an activity $a$ is determined by two events: leading event $e_i$, and ending event $e_j$ such that $e_i, e_j \in E$, $s \in S$, and $(e_i, s), (s, e_j) \in A$ of $P(a)$. To denote an event *name* of $P(a)$ we use $e_{name}(a)$. Table 6.2 presents an example of a few event names used to describe activity events[8]. We use $t(e)$ to refer to the timestamp of an event $e$ and $t_{now}$ to denote the timestamp at which the analysis is conducted. Because the monitoring and analysis is conducted at the runtime, it is possible that an activity $a$ is on a state $s$ but there is no such $(s, e) \in A$ of $P(a)$. When analyzing such state $s$, we use $t_{now}$ as a timestamp to determine the time spent on state $s$. We use $\rightarrow$ to denote the *happened before* relation between events.

The monitoring system collects states and events of each activity instance, and builds the execution graph of that activity instance. Currently, to get execution status of activities from WIC we manually instrument the WIC because WIC does not provide interface for the monitoring tool to obtain that information.

### 6.4.3 Intra-activity and Inter-activity Performance Metrics

Performance measurements for a Grid WF are collected at two levels: *activity* and *workflow* level. Based on monitoring data, performance measurements and WF graphs, the performance of WF is analyzed.

#### 6.4.3.1 Activity Level

At activity level, several performance metrics that characterize an activity are provided. We capture performance metrics of the activity, for example its execution status, performance measurements of code regions (e.g. wall-clock time, hardware metrics), etc. Firstly, we dynamically instrument code regions of the invoked application of the activity. We collect performance metrics such

[8] Detailed possible activity events can be found in [13].

as wall-clock time, CPU time and hardware counters of instrumented code regions. Performance metrics of code regions are incrementally provided to the user during the execution of the workflow. Based on these metrics, various data analysis techniques can be employed, e.g. load imbalance and metric ratio. We extend our overhead analysis for parallel programs (see Chapter 5) to WFAs. For each activity, we analyze *activity overhead*. Activity overhead contains various types of overhead, e.g. communication, synchronization, that occur in an activity instance.

Secondly, we focus on analyzing response-time of activities. *Activity response time* corresponds to the time an activity takes to be finished. The response time consists of waiting time and processing time. Waiting time can be queuing time, suspending/resuming time and processing time can consist of communication and computation time. For each activity $a$, its execution graph, $P(a)$, is used as the input for analyzing activity response time. Moreover, we analyze synchronization delay between activities. Let consider a dependency between two activities $(a_i, a_j)$ where $a_i \in pred(a_j)$. $\forall a_i \in pred(a_j)$, when $e_{completed}(a_i) \rightarrow e_{submitted}(a_j)$, the synchronization delay from $a_i$ to $a_j$, $T_{sd}(a_i, a_j)$, is defined as

$$T_{sd}(a_i, a_j) = t(e_{submitted}(a_j)) - t(e_{completed}(a_i)) \tag{6.1}$$

If at the time of the analysis $e_{submitted}(a_j)$ has not occurred, $T_{sd}(a_i, a_j)$ is computed as

$$T_{sd}(a_i, a_j) = t_{now} - t(e_{completed}(a_i)) \tag{6.2}$$

Each activity $a_j$ associates with a set of the synchronization delays. From that set, we compute maximum, average and minimum synchronization delay at $a_j$. Note that synchronization delay can be analyzed for any activity which is dependent on other activities. This metric is particularly useful for analyzing synchronization points in a workflow.

#### 6.4.3.2 Workflow level

We analyze performance metrics that characterize the interaction and the performance impact among activities. There are various metrics of interest such as average response time, waiting time, queuing time and synchronization delay of activities, load imbalance, computation to communication ratio, service requests per activities, activity usage, and success rate of activity invocation. Correlation metrics, such as number of activities per resource, resource utilization, etc., are also important.

We combine WF graph, execution status information, and performance data to analyze load imbalance for fork-join model. Let $a_0$ be the activity at the fork point. $\forall a_i, i = 1 : n, a_i \in succ(a_0)$, load imbalance $T_{li}(a_i, s)$ in state $s$ is computed as

$$T_{li}(a_i, s) = T(a_i, s) - \frac{\sum_{i=1}^{n} T(a_i, s)}{n} \tag{6.3}$$

We also apply load imbalance analysis to a set of selected activities. In a workflow, there could be several activities whose functions are the same, but are not in fork-join model. Load imbalance analysis is useful technique to reveal the work distribution.

### 6.4.4 Multi-workflow Analysis

We analyze *slowdown factor* for fork-join model. Slowdown factor, $sf$, is defined as

$$sf = n \times \frac{max_{i=1}^{n}(T_n(a_i))}{T_1(a_i)} \tag{6.4}$$

where $T_n(a_i)$ is the processing time of activity $a_i$ in fork-join version with $n$ activities and $T_1(a_i)$ is the processing time of activity $a_i$ in the version with single activity. We also extend the slowdown factor analysis to fork-join structures that contain structured block of activities. In this case, $T_n(a_i)$ will be the processing time of a structured block of activities in a version with $n$ blocks.

For different replaced refinement graphs of WFs of the same WFA, we compute *speedup* factor between them. Let $SG$ be a subgraph of workflow $WF_g$ of a WFA; $SG$ has $n_g$ components. Let $P_i =< a_{i1}, a_{i2}, \cdots, a_{in} >$ be a critical path from starting node to the ending node of the component $i$, $C_i$, of $SG$. The processing time of $SG$, $T_{cp}(SG)$, is defined as

$$T_{cp}(SG) = max_{i=1}^{n_g}(T_{cp}(C_i)), T_{cp}(C_i) = \sum_{k=1}^{n} T(a_{ik}) \tag{6.5}$$

where $T(a_{ik})$ is the processing time of activity $a_{ik}$. Now, let $SH$ be the replaced refinement graph of $SG$; $SG$ and $SH$ are subgraphs of workflow $WF_g$ and $WF_h$, respectively, of a WFA. Speedup factor *sp* of $SG$ over $SH$ is defined as follows:

$$sp = \frac{T_{cp}(SG)}{T_{cp}(SH)} \tag{6.6}$$

The same technique is used when comparing the speedup factor between two workflow $WF_g$ and $WF_h$.

In order to support multi-workflow analysis of WFs, we collect and store different DAGs, subgraphs of the WFA, performance data and machine information into an experiment repository powered by PostgreSQL. Each graph is stored with its associated performance metrics; graphs can be DAG of the WF or a subgraph. We use a table to represent refinement relationship between subgraphs. Currently, for each experiment, the user can select subgraphs, specifying refinement relation between two subgraphs of two WFs. The analysis service uses data in the experiment repository to conduct multi-workflow analysis.

## 6.5 Ontology-based Approach to Performance Analysis, Data Sharing and Tool Integration

The recent emerging Grid computing raises many challenges in the domain of performance analysis. One of these challenges is how to understand and utilize performance data where the data is diversely collected and no central component manages and provides semantics of the data. Performance monitoring and analysis on Grids differ from that in conventional parallel and distributed systems in terms of no single tool providing performance data for all Grid sites and the need of conducting monitoring, measurement and analysis across multiple Grid sites at the same time. Normally users execute their applications on multiple Grid sites, each is equipped with different computing capabilities, platforms and libraries, that require various tools to conduct performance monitoring and measurement. Without the central component, performance monitoring and measurement tools have to provide means for seamlessly utilizing the data they collect and provide because many other tools and services atop them need the data for specific purposes such as performance analysis, scheduling and resource matching. Current Grid performance tools focus on the monitoring and measurement, but neglecting the data sharing and tool integration.

In previous work, presented in Section 6.2, we have introduced a middleware for performance monitoring and data integration in Grids. In that middleware, we use XML to describe various types of performance data. It, however, turns out that XML does not provide mechanism to represent the semantics of such diverse performance monitoring in Grids. Encouraged by the Semantic Grid approach [72, 234], in order to overcome the limitation of XML we take a new direction in describing the semantics of performance data and establishing performance data sharing and tool integration by investigating the use of *ontology* in performance analysis domain. Basically, ontologies provide a *"shared and common understanding of a domain"* that can be used in the communication between people and application systems [116, 90]; ontology is developed to *"facilitate knowledge sharing and reuse"* [116, 90, 171]. Based on sharable and extensible ontologies in the domain of performance analysis, an analysis tool, service or user is able to access multiple sources of performance and monitoring data provided by a variety of performance monitoring and measurement tools, understanding the data and making use of that data. With the expressive power provided by ontology that can describe concepts, resources in sufficient detail, the chance of supporting automated performance analysis will also be increased.

### 6.5.1 Semantics of Performance Data

Most monitoring and measurement tools collect raw data (e.g. performance metrics) of monitored objects but they do not directly model and clarify the relevant aspects of monitored objects. As a result, the collected data lacks semantics and it is difficult to correlate the data with the appropriate domain knowledge. The software program hardly understands and handles that data, thus not fostering to automatically detect, correct and predict behavior of applications and computing systems at runtime. Lack of detailed model of monitored resources also prevents us from moving performance analyzers as close to the monitored source as possible. As a result, the performance analysis for the Grid is still conducted in the centralized manner even though the monitoring and measurement are conducted in the distributed manner.

Currently, several data representations with different capabilities and expressiveness are employed by Grid performance monitoring and measurement tools such as XML [270], XML schema [271] and relational database schema. However, little effort has been done to standardize the semantics of performance data as well as the way the performance tools collaborate each other. In the Grid, data is diversely collected and no central component manages and provides its semantics. Each Grid site may be equipped with its own performance monitoring and measurement tool. Thus, the end user or the high level tool in Grids has to interact with a variety of tools offering monitoring and measurement service. Performance monitoring and measurement tools should not simply offer well-defined operations for other services to call them (e.g. based on OGSA [93, 94]) but they have to provide means for adding semantics to the data as Grid users and services require seamless integration and utilization (performance) data provided by different tools.

Existing approaches for performance data sharing and tool integration which mostly focus on building wrapper libraries for directly converting data between different formats, storing data in relational database with specific data schema, or exporting data into XML, have several limitations. For example, based on our experiences on developing data integration (see Section 6.2.3.8), building a wrapper requires high implementation and maintenance costs; wrappers convert data between representations but not always between semantics.

Although XML and XML schemas are sufficient for exchanging data between parties that have agreed in advance on definitions, use and meaning of XML vocabularies, they mostly are suitable for one-to-one communication. XML only provides a syntax for encoding structured documents and imposes no semantic constraints on the meaning of these documents. XML schema imposes constraints and restrictions on the structure of XML documents, but not on the semantics, and extends XML with data types. Everyone can create his own XML vocabularies with his own definitions for describing his data. However, such vocabularies and definitions are not sharable and do not establish a common understanding about the data, thus preventing semantic interoperability between various parties which is an important issue that Grid monitoring and measurement tools have to support. Utilizing relational databases to store performance data [245, 235] makes data more sharable and accessible. However, the data models represented by relational database are still very tool-specific and inextensible. Integrating different relational tables by using SQL join statements may generate nonsense information as relational database checks the data type, instead of the semantics, of the data. Moreover, relational database is not extensible. Extending these database schemas to cover new performance metrics or monitored objects would result in daunting tasks of redesigning relations and attributes. Notably, XML and relational database schemas do not explicitly express meanings of data they encode. Since all above-mentioned techniques do not provide enough capability to express the semantics of performance data and to support tool integration, they might not be suitable for describing performance data in the Grid due to the autonomy and diversity of performance monitoring and measurement tools.

We investigate whether the use of ontology can help to solve the above-mentioned issues. Ontologies are a popular research topic in various communities such as representation and reasoning, information integration and cooperative information systems [90]. Ontologies were originally introduced by the artificial intelligence community to *"facilitate knowledge sharing and reuse"* [116, 90, 171] and recently ontology has been considered as the main tool that can be used to achieve the semantic interoperability in the Grid [72, 234]. A short overview of ontology and ontology languages can be found in Appendix G.

### 6.5.2 Using Ontology to Describe Performance Data

There are many ways of using ontology for addressing the issues mentioned in Section 6.5.1. Firstly, ontology can be used to directly *describe and model the data* collected, thus allows performance tools to share a common understanding of performance data and to correlate the data with the knowledge domain. Secondly, ontology can be used to *define mappings between different representations* employed by different Grid monitoring and measurement tools. In both cases, we can utilize the single ontology or hybrid ontology approach [214]. In the single ontology approach, we define a single ontology for performance analysis domain. In the latter approach, each performance tool can define its own ontology based on a shared ontology. This would allow a high level service to transparently access different types of data in a homogeneous way.

### 6.5.3 PERFONTO: Ontology for Describing Performance Data

While ontology has widely been applied successfully to represent data in many fields such as AI, Semantic Web, Health and Biology [9], before starting working on this topic, we were not aware of such an ontology for performance data in the field performance analysis. Our initial effort is that we propose an ontology for representing performance data in the Grid with the hope that the proposed ontology will not only serve for data sharing and reuse between

[9] For example, refer to http://www.daml.org/ontologies/category.html

performance analysis tools but also increase the automation of performance analysis process. In this section, we describe PERFONTO, an ontology for describing performance data that can be used by various performance monitoring and measurement tools.

#### 6.5.3.1 Selecting Ontology Language

Although various languages can be used to represent ontology such as RDFS [213], DAML [158], OIL [161], DAML+OIL [71], we choose OWL (WebOnt) [188] for describing performance ontology due to several reasons. OWL is developed as a vocabulary extension of RDF [205] and is derived from DAML+OIL. OWL can describe classes and properties in more complex ways than RDFS. For example, properties in RDFS can be organized into a property hierarchy (one property may be a subproperty of another). In OWL, properties can be denoted as transitive, symmetric or functional, and one property can be defined as an inverse of as well as a subproperty of another. OWL distinguishes two types of properties: *Data Property* and *Object Property*. Data properties have data-values as their range whereas object properties have individual-values as their range. OWL is currently being standardized by W3C [239].

OWL follows the object-oriented approach, with the structure of the domain being described as a set of definitions of *classes* and *properties*. OWL consists of a set of *axioms* including *class axioms* and *property axioms* that assert subsumption relationships between classes and properties. Class axioms specify necessary and/or sufficient characteristics of a class, e.g., sub class and equivalent class, whereas a property axiom defines characteristics of a property such as range, domain and relations to other properties.

#### 6.5.3.2 PERFONTO Design

PERFONTO comprises two parts that describe *experiment-related concepts* and *resource-related concepts*. Here we briefly discuss main classes in current version of PERFONTO.

Experiment-related concepts describe experiments and their associated performance data of applications. The structure of the concepts is described as a set of definitions of *classes* and *properties*. Figure 6.24 presents descriptions of part of experiment-related classes in PERFONTO. `Application` describes information about the application. `Version` describes information about versions of an application. `SourceFile` describes source file of a version. `CodeRegion` describes a static (instrumented) code region. Code regions are classified into sub-classes that are programming paradigm-dependent (e.g. message passing and shared memory code regions) and paradigm-independent (e.g. loops and arbitrary code regions).

`Experiment` describes an experiment which refers to a sequential or parallel execution of a program. `RegionInstance` describes a region instance which

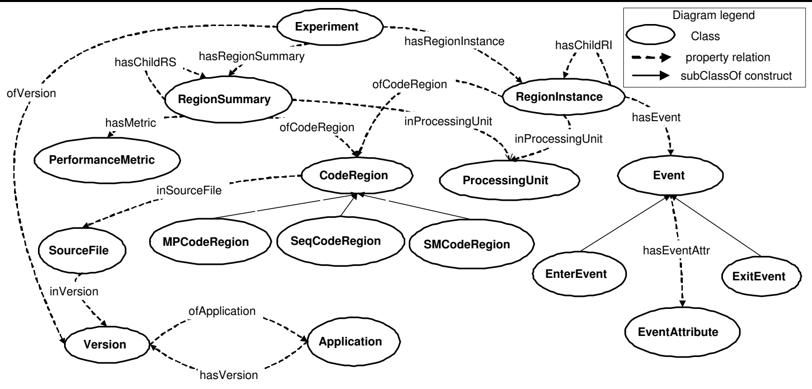


**Fig. 6.24.** Illustrative classes and properties of experiment-related concepts.

is an execution of a static (instrumented) code region at runtime. A dynamic code region instance is associated with its calling context in a processing unit (property `inProcessingUnit`) and has events (`hasEvent`) and sub region instances (`hasChildRI`). `RegionSummary` describes the summary of dynamic code region instances of a static (instrumented) code region in a processing unit. A region summary has performance metrics (`hasMetric`) and sub region summaries (`hasChildRS`). `PerformanceMetric` describes a performance metric, each metric has a name and value (`hasMetricName, hasMetricValue`). `Event` describes an event record. Sub classes of `Event` are `EnterEvent` and `ExitEvent` that describe enter and exit event of a region instance, respectively. An event happens at a time (`atEventTime`) and has event attributes (`hasEventAttr`). `EventAttribute` describes an attribute of an event which has an attribute name and value (`hasAttrName, hasAttrValue`). An excerpt of OWL for `RegionSummary` is shown in Figure 6.25.

Resource-related concepts describe static, benchmarked, and dynamic (performance) information of computational nodes and networks. In the current version, resource-related concepts provide classes to describe static and benchmarked data of computational and network resources. For example, `Site` describes information of a Grid site. `Cluster` describes a set of physical machines (computational nodes). Cluster may have further subclasses such as `SMPCluster` represented a cluster of SMP. `ComputationalNode` describes information about physical machine. `ComputationalNode` may have other subclasses such as `SMPComputationalNode` represented an SMP machine. `Network` describes an available network. Further sub classes of Network can be `EthernetNetwork`, `MyrinetNetwork`, etc. `NodeSharedMemoryPerf` describes performance characteristics of shared memory operations of a computational node. `NetworkMPColPef` and `NetworkMPP2PPerf` describe performance characteristics of message passing collective and point-to-point operations of a network, respectively. Other classes describing performance of high level protocols (e.g., HTTP) are currently being investigated.

```
<owl:Class rdf:ID="RegionSummary">
     <rdfs:label>RegionSummary</rdfs:label>
     <rdfs:comment>Class represents for RegionSummary</rdfs:comment>
</owl:Class>
<owl:ObjectProperty rdf:ID="inProcessingUnit">
     <rdfs:label>inProcessingUnit</rdfs:label>
     <rdfs:comment>information about processing unit</rdfs:comment>
     <rdfs:domain rdf:resource="#RegionSummary"/>
     <rdfs:range rdf:resource="#ProcessingUnit"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="ofCodeRegion">
     <rdfs:label>ofCodeRegion</rdfs:label>
     <rdfs:comment>represents code region information</rdfs:comment>
     <rdfs:range rdf:resource="#CodeRegion"/>
     <rdfs:domain rdf:resource="#RegionSummary"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasChildRS">
     <rdfs:label>hasChildRS</rdfs:label>
     <rdfs:comment>list of child region summaries</rdfs:comment>
     <rdfs:domain rdf:resource="#RegionSummary"/>
     <rdfs:range rdf:resource="#RegionSummary"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasMetric">
     <rdfs:label>hasMetric</rdfs:label>
     <rdfs:comment>list of performance metrics</rdfs:comment>
     <rdfs:range rdf:resource="#PerformanceMetric"/>
     <rdfs:domain rdf:resource="#RegionSummary"/>
</owl:ObjectProperty>
```

**Fig. 6.25.** Description of `RegionSummary` has four object properties namely `inProcessingUnit, ofCodeRegion, hasChildRS, hasMetric` that specify associated processing unit, code region, sub region summary and performance metric of the region summary, respectively.

The proposed ontology is largely based on our previous work on developing data schema for expressing experiment data in relational database (presented in Section 5.5) and monitoring data in XML (presented in Section 6.2), and on APART experiment model [83]. Our proposal cannot cover all current representations of existing performance tools. Rather, it is the first step to establish an ontology for performance data that tries to describe semantics of performance data currently supported by most existing performance monitoring and measurement tools.

#### 6.5.3.3 PERFONTO for Information Integration

One of the main key advantages of ontology is that the semantics is explicitly defined and different ontologies can be reconciled [169, 183]. In fact, PERFONTO can be considered as a merging of two different ontologies. Suppose

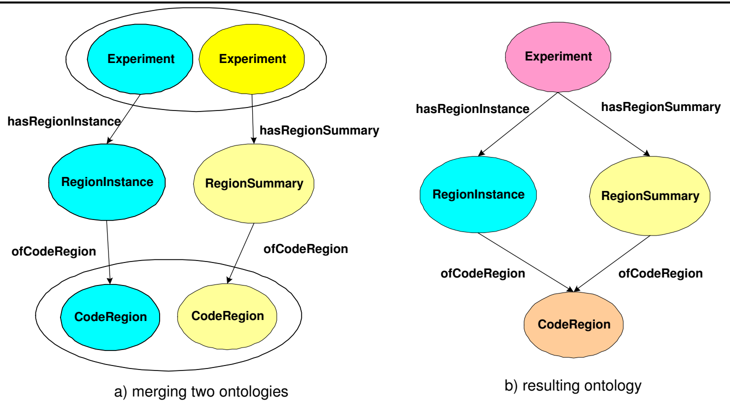


**Fig. 6.26.** An example of merging two ontologies. Two ontologies share the same concepts of *Experiment* and *CodeRegion* which can be merged.

a monitoring tool $A$ provide only tracing information. Therefore, tool $A$ does not employ `RegionSummary` concept, which mostly is used to describe profiling data. On the other hand, a monitoring tool $B$ supports profiling measurement thus $B$ does not use `RegionInstance` concept, which mostly is used to describe trace events. However, both $A$ and $B$ employ other concepts in PERFONTO such as `Experiment`, `CodeRegion`. Another tool $C$ and a performance analysis service $S$ want to deal with both profiling and tracing provided by $A$ and $B$. $C$ and $S$ can employ PERFONTO which is a result from a merging of the two ontologies developed by $A$ and $B$. Figure 6.26 presents the above merging example.

The development of PERFONTO should be considered as the investigation of using ontology for describing performance data, not establishing a standard for all tools. PERFONTO is designed not only for describing performance data but also for serving as a shared ontology for developing other tool-specific ontologies. Therefore, one can employ or extend PERFONTO for representing his performance data. Others may develop their own ontologies. However, all proposed ontologies can be merged. As a result, the way to share and reuse performance data will not be hampered.

### 6.5.4 Architecture of Ontology-based Performance Analysis, Data Sharing and Tool Integration

Based on PERFONTO, we can share common understanding of the performance data structures among performance tools and high-level tools. Therefore, performance data can easily be understood and reused by various services. Figure 6.27 presents a three layers architecture of ontology-based performance analysis, data sharing and tool integration. The core of this architecture is an ontology-based performance data repository service which includes:

- *PERFONTO* is the ontology for describing performance data discussed in Section 6.5.3.
- *Ontological database* is a relational database used to hold ontologies (e.g PERFONTO) and performance data (instance data).
- *PERFONTO APIs* are interfaces used to store and access data in ontological database.
- *Query Engine* provides searching and querying functions on ontological data.
- *Inference Engine* provides reasoning facilities to infer, discover and extract knowledge on ontological performance data.

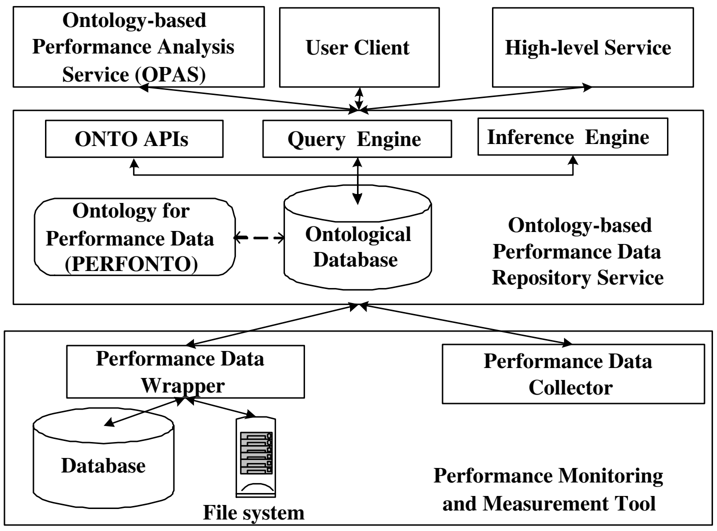

Ontology-based Performance Analysis Service (OPAS)
User Client
High-level Service
ONTO APIs
Query Engine
Inference Engine
Ontology for Performance Data (PERFONTO)
Ontological Database
Ontology-based Performance Data Repository Service
Performance Data Wrapper
Performance Data Collector
Database
File system
Performance Monitoring and Measurement Tool

**Fig. 6.27.** Three layers architecture for ontology-based performance analysis, data sharing and tool integration

The ontology-based repository is designed as a Grid service. The *Performance Data Collector* of performance monitoring and measurement tools can store collected data (instance data) along with corresponding ontology model (e.g. PERFONTO) into the ontological repository. Via service operations, any clients which need performance data such as performance analysis tools, schedulers or users can easily request and retrieve the ontology model and instance data from ontological database. The key difference from approaches of using XML or relational database is that performance data is either described by a common ontology (e.g. PERFONTO) or by a tool-defined ontology. Therefore, with the presence of ontology model, these clients can easily understand and automatically process the retrieved data. Through *Performance Data*

*Wrapper*, data in tool-defined non-ontology format can also be extracted and transformed into ontological representation and vice versa.

Figure 6.28 presents client-server interaction model between the ontology-based performance data repository service and its client in which clients connect to instances of the service in sites, retrieving data from these instances and making use of that data locally. Moreover, clients can invoke search and inference operations on remote data repository.

Performance data of experiments in distributed systems generally, Grid systems particularly, may be stored in different repository services, for instance in three sites $VO_A$, $VO_B$ and $VO_C$ shown in Figure 6.28. The ontological repository service discussed above does not deal with the issue of managing that distributed data as it just provides a storage for archiving data. That issue is well-known in distributed database management [184]. At the meantime, we employ a simple scheme assuming that the experiment planer (e.g., OPAS and user) which runs experiments controls and manages information about experiments in the Global Experiment Metadata (GEM). Information about an experiment includes a unique experiment identifier, location of repository services that hold performance data, etc. Clients use information in GEM to access application performance data stored in multiple repository services.

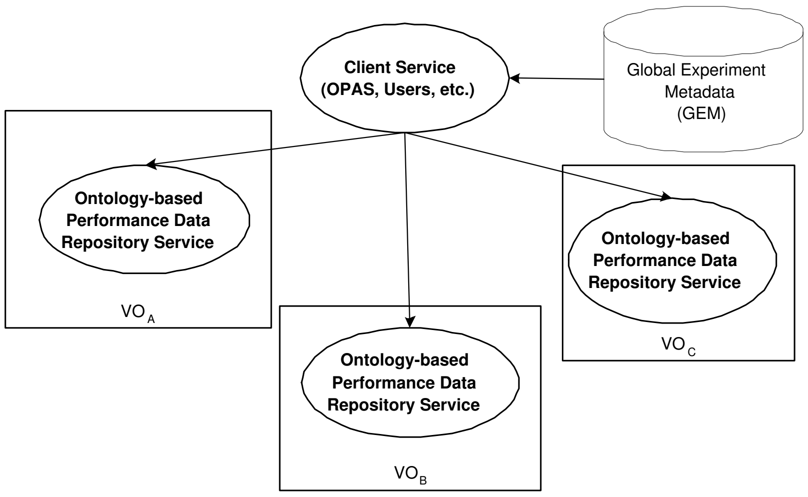

Client Service (OPAS, Users, etc.)
Global Experiment Metadata (GEM)
Ontology-based Performance Data Repository Service
VO_A
Ontology-based Performance Data Repository Service
VO_B
Ontology-based Performance Data Repository Service
VO_C

**Fig. 6.28.** Model of interactions between clients and ontology-based performance data repository service.

To implement this architecture we select Jena [138] for processing ontology-related tasks. Jena is an open source and grown out of work with the HP Labs Semantic Web [130]. It is an Java framework for building Semantic Web applications and provides a programming environment for RDF, RDFS and OWL, including a rule-based inference engine. The ontology-based performance data repository service is developed based on OGSA [93, 94]. In the following we detail components of ontology-based performance data repository.

#### 6.5.4.1 Ontological Database

Ontological database is used as a persistent storage for ontology descriptions and instance data. Given an application, for each experiment, performance data is a set of instances (individuals) following a specific ontology model (e.g. PERFONTO). Similarly, for each Grid site system performance data are collected and archived. We can store both ontology model and instance data in persistent storage. Note that the ontological database is used as a storage. The task of storing and retrieving ontological data is done by other components (e.g., sensors and consumers)

We use a table named `ExperimentData (experimentName, descriptionModelName, instanceModelName)` to keep relation between experiments, ontology model and instance data of experiments locally stored in this database. To store ontology model and instance data into persistent storage we use Jena API for persistent database models. The ontology model and associated instance data are kept separately although this is not required. By doing so, one ontology model can efficiently be applied to several sets of instance data. By connecting to the service, clients can obtain information in table *ExperimentData*. Based on that information, clients can retrieve entirely or partially any available ontology models, instance data and/or perform operations on these models such as search and reasoning.

#### 6.5.4.2 Search on Ontological Data

A search engine can be developed to support the user on finding interesting data in the ontological database. At the initial step, we use a search engine provided by Jena. The search engine supports RDQL query language [159, 216] which is an implementation of an SQL-like query language for RDF. Jena search engine [138] treats RDF as data, provides query with triple patterns and constraints and returns a list of bindings; each binding is a set of name-value pairs for the values of the variables.

RDQL syntax is similar to SQL. In RDQL, the `SELECT` clause identifies the variables to be returned to the application; variables are introduced with a leading '?'. The `FROM` clause specifies the model by URI whereas the `WHERE` clause specifies the graph pattern as a list of triple patterns. The `AND` clause specifies the Boolean expressions. RDQL introduces `USING` clause providing a way to shorten the length of URIs.

The use of RDQL in combining with ontology can simplify and provide a *high level model of search* in performance analysis in which search queries can be easily understood and defined by end-user, not only by the tool developer. Let us show a simple example: a client wants to find any region

```
l1:  SELECT ?regionsummary
     WHERE
l2:       (?regionsummary perfonto:inProcessingUnit ?processingunit)
l3:       (?processingunit perfonto:inNode "gsr410")
l4:       (?regionsummary perfonto:hasMetric ?metric)
l5:       (?metric perfonto:hasMetricName "wtime")
l6:       (?metric perfonto:hasMetricValue ?value)
l7:  AND (?value >=3E8)
l8:  USING perfonto
     FOR <http://www.par.univie.ac.at/project/scalea/perfonto#>
```

**Fig. 6.29.** An example of RDQL query based on PERFONTO.

summary executed in computational node `gsr410` with wall-clock time (denoted by metric name `wtime`) greater than or equal to `3E8` microseconds. The corresponding RDQL query based on PERFONTO is presented in Figure 6.29. Line $l_1$ selects variable `regionsummary` via `SELECT` clause. In line $l_2$ information about processing unit of `regionsummary`, determined by property `perfonto:inProcessingUnit`, is stored in variable `processingunit`. The computational node of `processingunit` must be `‘‘gsr410’’` as stated in line $l_3$. In line $l_4$, performance metric of `regionsummary` is stored in variable `metric` and line $l_5$ states that the name of `metric` must be `‘‘wtime’’`. In line $l_6$, the value of `metric` is stored in variable `value` which must be greater than or equal to 3E8 as specified in line $l_7$. Line $l_8$ specifies the URI for the shortened name `perfonto`.

#### 6.5.4.3 Reasoning on Ontological Data

The use of ontology for representing performance data allows additional facts to be inferred from instance data and ontology model by using axioms or rules. Based on ontology, we can employ inference engine to capture knowledge via rules.

Let us analyze a simple rule for detecting all MPI point-to-point communication code regions whose the average message length is greater than a predefined threshold [83]. As presented in Figure 6.30, line $l_1$ defines the name of the rule. In line $l_2$, a term of triple pattern specifies link between a region summary and its associated code region. Line $l_3$ states that the code region is an instance of `MPCodeRegion` (message passing code region) and is an MPI point-to-point communication region (denoted by mnemonic CR_MPIP2P) as specified in line $l_4$. Line $l_5$, $l_6$ and $l_7$ are used to access the average message length of the region summary. Line $l_8$ checks whether the average message length is greater than a predefined threshold `BIG_MESSAGES_THRESHOLD` by using a built-in function. In line $l_9$, the action of this rule concludes and prints the region summary having big message. This example shows how using ontology helps simplifying the reasoning on performance data.

```
[rule_detect_bigmessages:
     (?regionsummary perfonto:ofCodeRegion ?codeRegion),
     (?codeRegion rdf:type perfonto:MPCodeRegion),
     (?codeRegion perfonto:hasCrType "CR_MPIP2P"),
     (?regionsummary perfonto:hasMetric ?metric),
     (?metric perfonto:hasMetricName "AvgMessageLength"),
     (?metric perfonto:hasMetricValue ?length),
     greaterThan(?length, BIG_MESSAGES_THREADHOLD)
->       print(?regionsummary,"Big message hold!")]
```

**Fig. 6.30.** An example of Rule-based Reasoning based on PERFONTO.

#### 6.5.4.4 Converting Performance Data

Even with the presence of an ontology for performance data, it does not mean that existing tools will always store their data into ontological representation. Firstly, not all data needs to be shared. Secondly, expressing performance data in ontological representation requires more space to store. To minimize archival space, tools may only store portion of performance data on ontological representation when needed.

Existing tools can convert performance data into ontology-based representation. This data can be stored into files, relational-database and easily be accessed via PERFONTO APIs as well as Query Engine. A wrapper needs to be developed in order to extract/transform performance data between two representations: the tool internal representation and the ontological representation.

### 6.5.5 Discussion on Enhanching Automatic Performance Analysis

An automatic performance analysis system consists of a large components based on different tools and specifications [105]. The use of PERFONTO particularly and ontology generally can help increasing the automation of performance analysis. For example, to automate basic performance analysis experiments, performance analyzers must be able to process information (including static and dynamic) about the application collected, mostly, by different tools. The idea of using ontology (e.g. PERFONTO) fits well for that purpose as ontology can semantically describe the information about applications. Moreover, by using ontology, high-level components such as analysis agents [103] are able to understand data provided by many sources in heterogeneous environment even if they do not know the exact type of that data in advance. Another potential of ontology that can be applied in automatic performance analysis is reasoning capability. As simply demonstrated in Section 6.5.4.3, we can define rules for automatic performance analysis. Based on predefined patterns, the action of rules can automatically determine and perform appropriate tasks without or with little any human intervention. By doing so, we can also move analysis components as close to analyzed resources as possible.

Ontology can also be used for ensuring semantically interactions between tools of an automatic performance analysis system such as instrumentation systems, moniotoring services and performance analysis services. The ontology can represent features of available tools by classifying their main components and specifying the relationships and contraints among them, thus allowing users/services to select and configure the most suitable solution for automatically executing a performance analysis process.

## 6.6 Summary

For monitoring and analyzing performance of applications in the Grid, we need to collect and process a variety of types of performance data from many sources in a uniform way. The sensor-based middleware for performance monitoring and data integration in Grids is developed for that purpose. The middleware unifies different types of sensors for monitoring and performance analysis in a single system and integrates various types of data from many sources. To cope with the diversity and dynamics of the Grid, Grid-based service operations and TCP-based data stream are employed in order to balance the tradeoffs between the interoperability and manageability and the performance. Our middleware stores collected data in decentralized locations and provides the same mechanism for accessing that distributed data. By incorporating P2P and autonomic technologies, the middleware is capable of self-management.

We have described a novel Grid service for supporting dynamic instrumentation of Grid applications. The dynamic instrumentation service offers a widely accessible interface, through Web/Grid services, and highly interoperable protocols named IRL and SIRBC, for other services to conduct the instrumentation. A Grid service for performance analysis of scientific workflows in the Grid has been developed. The analysis service employs the monitoring middleware and the dynamic instrumentation service, utilizes various types of performance data including resource monitoring, execution status of activities, and performance measurement collected from instrumented applications, and supports online monitoring and performance analysis of scientific workflows. Novel techniques to support multi-workflow analysis are introduced.

A new approach to performance monitoring, data integration and analysis that is based on ontology is introduced. Ontology is used to represent performance data in order to make sure that different services have a common understanding of performance data provided by diverse sources. Using the approach based on ontology, we can overcome the lack of semantics in current data representations. Performance data will be easily shared and processed by automated tools, services and human users, thus leveraging the data sharing and tool integration, and enhancing the automatic performance analysis.

# 7 Experiments

## 7.1 Introduction

In this chapter we give an overview of the current implementation of our performance measurement, instrumentation and analysis systems. We then present experiments with our techniques and methods. First, we present performance analysis experiments of a variety of parallel applications, covering OpenMP, MPI, HPF and mixed parallel programs, for the cluster architecture. Second, we present experiments of monitoring, instrumentation and performance analysis conducted on the Grid.

## 7.2 Implementation Overview

### 7.2.1 SCALEA

SCALEA as shown in Fig. 4.1 has been fully implemented. However, some performance overheads (see Fig. 5.1) which include replicated code, algorithm change, compiler change, implicit barrier operations, scheduling, and communication overhead of reduction operation, are not yet supported.

SIS is implemented based on VFC compiler. Currently, SIS can be run on Sun Solaris platforms only. SISPROFLING provides libraries for measuring C/C++ and Fortran programs executed on various Unix and Linux platforms, but not on Windows. The experiment data repository is implemented with PostgreSQL [10] which is a relational database system supported on many platforms. All components interacting with the data repository are written in Java and the connection between these components with the database is powered by JDBC. This approach has the advantages of portability and network capability. For instance, analysis components can be run on Window machines supporting Java while the data repository can be setup on a different UNIX machine.

We implement soft performance analysis techniques in SCALEA framework. We use the NRC-IIT FuzzyJ Toolkit [101] from the National Research Council of Canada's Institute for Information Technology as the base library for fuzzy computing.

### 7.2.2 SCALEA-G

Most components of SCALEA-G are written in Java except the application sensor library and the Mutator Service are written in C/C++. Globus Toolkit (GT) [108] and JavaCog [160] are used as base libraries for implementing OGSA-enabled services, communication and security in SCALEA-G services. Currently we use GT version 3.2.

The library of application sensors utilizes Globus IO library for implementing communication and security tasks between the sensors and Sensor Manager Service. The facilities for dynamic instrumentation of applications at runtime are provided by Dyninst [51]. The Grid dynamic instrumentation service is implemented based on gSOAP, a C++ Web Service toolkit with GSI-plugin [117].

PostgreSQL [10] has been used for archiving information in the directory service. To buffer the data at Sensor Manager Services, we use Berkeley DB provided by Sleepycat [11].

The workflow invocation and control (WIC) service in our experiment is currently implemented based on JavaCog [160]. JGraph [140] is used to visualize workflow DAGs. To visualize performance monitoring data and results we use JFreeChart [139] and ASKALON Visualization [27]. We store performance data of workflows into a repository based on PostgreSQL.

### 7.2.3 Ontology-based Performance Analysis

We have implemented a prototype of the proposed ontology and ontology-based service discussed in Section G. At each Grid site, we use SCALEA/ SCALEA-G to instrument applications. Performance measurements collected from instrumented programs are stored into a ontological database. We use existing benchmarks to obtain reference values of system performance for message passing (e.g. blocking send/receive, collective) and shared memory operations (e.g. lock, barrier) of computational nodes and networks. These values are then stored into ontological database. The ontological database has been developed based on PostgreSQL and accessed by libraries based on Jena [138]. Moreover, we developed a wrapper to convert existing performance data in SCALEA experiment database to the ontology-based representation.

The ontology-based performance data repository is implemented as an OGSA service and deployed at each Grid site. However, in current prototype this service supports only operations for retrieving and storing ontology descriptions and instance data; search and reasoning have to be done at the client side. We are working on providing service operations for search and reasoning.

We develop OPAS (Ontology-based Performance Analysis Service) to support ontology-based search and reasoning; OPAS which is a client of the repository service is based on client-server model presented in Figure 6.28. The result of RDQL returned by Jena is stored in triple forms. We have further developed visualization components to process triple forms, displaying the result in various forms such as tree, table. However, ontology rule-based inference for performance analysis has not been implemented. Meanwhile, to support rule-based inference, we focus on developing a set of analysis rules for discovering knowledge in ontological performance data.

## 7.3 Performance Analysis of Parallel Programs

In this section, we present experiments with SCALEA. Unless stated otherwise, experiments are conducted on an SMP cluster named Gescher [230]. The cluster consists of 16 nodes; each comprises 4 Intel Pentium III Xeon 700 MHz CPUs with 1MB full-speed L2 cache, 2Gbyte ECC RAM, Intel Pro/100+Fast Ethernet, Ultra160 36GB hard disk is run with Linux. We use `pgf90` compiler from the Portland Group Inc. We use MPICH [114]. Over the course of experiments, the operating system and software versions are frequently updated.

### 7.3.1 Molecular Dynamics (MD) Application

The MD program implements a simple molecular dynamics simulation in continuous real space. This program, obtained from [266], has been implemented as an OpenMP program.

Our experiments have been conducted on a SMP node which is run with Linux 2.2.18-SMP patched with `perfctr` for hardware counters performance. We use `pgf90` compiler version 3.2 from the Portland Group Inc.

| **Overhead** | **2CPUs** | **3CPUs** | **4CPUs** |
|---|---|---|---|
| Loss of parallelism | 0.025 | 0.059 | 0.066 |
| Control of parallelism | 1.013 | 0.676 | 0.517 |
| Synchronization | 1.572 | 1.27 | 0.942 |
| $T_i$ | 2.61 | 2.009 | 1.527 |
| $T_u$ | 0.855 | 0.903 | 0.908 |
| $T_o$ | 3.466 | 2.913 | 2.435 |
| Total execution time | 146.754 | 98.438 | 74.079 |

**Table 7.1.** Overheads (s) of the MD application. $T_i$, $T_u$, $T_o$ are identified, unidentified and total overhead, respectively.

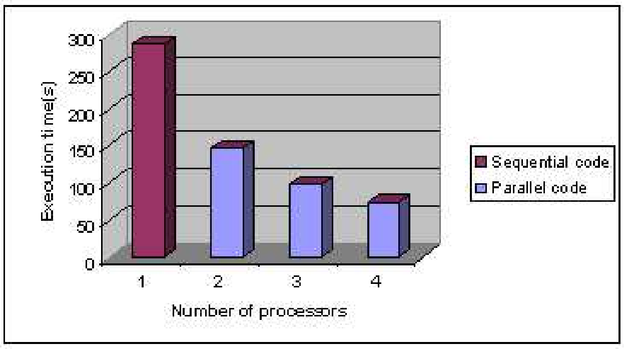


**Fig. 7.1.** Execution time of the MD application

The performance of the MD application has been measured on a single SMP node of Gescher. Figure 7.1 and Table 7.1 show the execution time behavior and measured overheads, respectively. The results demonstrate a good speedup behavior (nearly linear). As shown in Table 7.1, the total overhead is very small and large portions of the temporal overhead can be identified.

The time of the sequential code regions (unparallelized) does not change as it is always executed by only one processor. By increasing $p$ it can be easily shown that the loss of parallelism increases as well (see Remark 1, Section 5.2.5.1), which is also confirmed by the analysis results shown in Table 7.1.

Control of parallelism – mostly caused by loop scheduling – actually decreases when increasing the number of processors. A possible explanation for this effect can be that, for larger number of processors, the master thread processes less loop scheduling phases than for a smaller number of processors. The load balancing improves by increasing the number of processors/threads in one SMP node, which at the same time decreases synchronization time.

We then examine the cache miss ratio – defined by the number of L2 cache misses divided by the number of L2 cache accesses – of the two most important OMP DO code regions namely `OMP DO COMPUTE` and `OMP DO UPDATE` as shown in Figure 7.2. This ratio is nearly 0 when using only a single processor which implies very good cache behavior for the sequential execution of this code. All data seem to fit in the L2 cache for this case. However, in a parallel version, the cache miss ratio increases substantially as all threads process data of global arrays that are kept in private L2 caches. The cache coherency protocol causes many cache lines to be exchanged between these private caches which induces cache misses. It is unclear, however, why the master thread has a considerably higher cache miss ratio then all other threads. Overall, the cache behavior has very little impact on the speedup of this code.

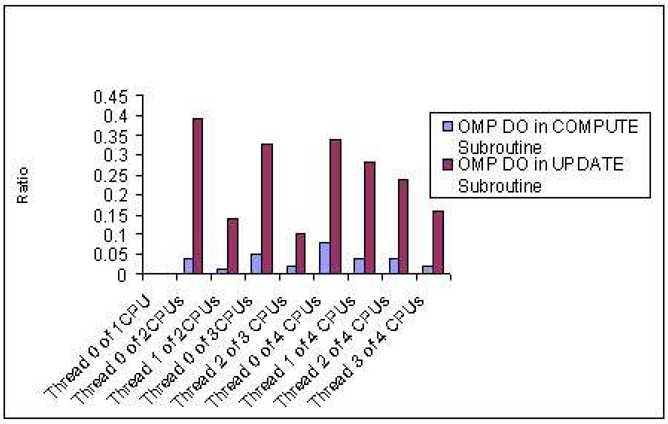


**Fig. 7.2.** The L2 cache misses/cache accesses ratio of OMP DO regions in the MD application.

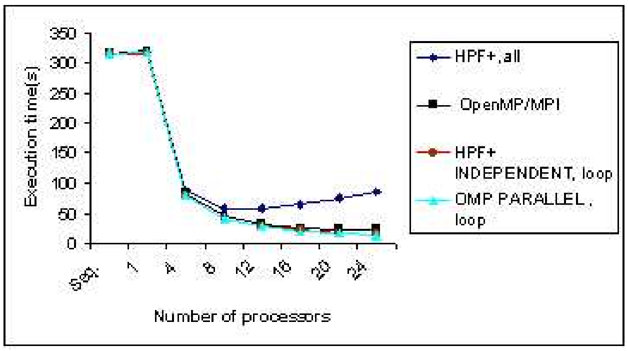


**Fig. 7.3.** Execution times of the HPF+ and OpenMP/MPI version for the backward pricing application

### 7.3.2 Backward Pricing Application

The backward pricing code [78] implements the backward induction algorithm to compute the price of an interest rate dependent financial product, such as a variable coupon bond. Two parallel code versions have been created. First, an HPF+ version that exploits only data parallelism and is compiled to an MPI program, and second, a mixed version that combines HPF+ with OpenMP. For the latter version, VFC generates an OpenMP/MPI program. HPF+ directives are used to distribute data onto a set of SMP nodes. Within each node an OpenMP program is executed. Communication among SMP nodes is realized by MPI calls. Our experiments have been conducted on SMP nodes of Gescher; node is run with Linux 2.2.18-SMP patched with `perfctr` for hardware counters measurement. We use MPICH 1.2.1 and `pgf90` compiler version 3.2 from the Portland Group Inc.

The execution times for both versions are shown in Figure 7.3. The term `all` in the legend denotes the entire program, whereas `loop` refers to the main computational loops (HPF INDEPENDENT loop and an OpenMP parallel loop for version 1 and 2, respectively). The HPF+ version performs worse than the OpenMP/MPI version which shows almost linear speedup for up to 2 nodes (overall 8 processors). Tables 7.2 and 7.4 display the overheads for the HPF+ and mixed OpenMP/MPI version, respectively. In both cases the largest overhead is caused by the control of parallelism overhead which rises significantly for the HPF+ version with increasing number of nodes. This effect is less severe for the OpenMP/MPI version. In order to find the cause for the high control of parallelism overhead we use SCALEA to determine the individual components of this overhead (see Tables 7.3 and 7.5). Two routines (`Update_HALO` and `MPI_Init`) are mainly responsible for the high control of parallelism overhead of the HPF+ version. `Update_HALO` updates the overlap areas of distributed arrays which causes communication if one process requires data that is owned by another process in a different node. `MPI_Init` initializes the MPI runtime system which also involves communication. The HPF+ version implies a much higher overhead for these two routines compared to the OpenMP/MPI version because it employs a separate process on every CPU of each SMP node, whereas the OpenMP/MPI version uses one process per node.

| **Processors** | **1N, 1P** | **1N, 4P** | **2N, 4P** | **3N, 4P** | **4N,4P** | **5N,4P** | **6N,4P** |
|---|---|---|---|---|---|---|---|
| $T_{odata}$ | 0 | 0.012 | 0.03 | 0.0207 | 0.0233 | 0.03028 | 0.0353 |
| $T_{octrl}$ | 0.244258 | 6.59928 | 17.2419 | 28.9781 | 41.4966 | 56.4554 | 70.7302 |
| $T_i$ | 0.244258 | 6.611285 | 17.2719 | 28.9988 | 41.5199 | 56.48576 | 70.76559 |
| $T_u$ | 3.139742 | 1.726465 | 1.835957 | 2.047059 | 2.3549 | 2.99739 | 2.5173 |
| $T_o$ | 3.384 | 8.33775 | 19.10787 | 31.0459 | 43.8749 | 59.48315 | 73.2829 |
| $T_p$ | 319.801 | 87.442 | 58.66 | 57.414 | 63.651 | 75.304 | 86.467 |

**Table 7.2.** Overheads (s) of the HPF+ version for the backward pricing application. $T_{odata}$ and $T_{octrl}$ are data movement overhead and control of parallelism overhead, respectively. $T_i$, $T_u$, $T_o$ are identified, unidentified and total overhead, respectively. $T_p$ is total execution time. The execution time of the sequential version, $T_s$, is 316.417 (s). *1N, 4P* means 1 SMP node with 4 processors.

### 7.3.3 Stommel Model of Ocean Circulation Application

Our experimental code represents a mixed OpenMP/MPI Fortran program that solves the 2d Stommel Model of Ocean Circulation using a Five-point stencil and Jacobi iteration. This code has been automatically instrumented, executed, measured, and analyzed for several problem and machine sizes based on user-provided SIS directives inserted in the source code. Our experiments have been conducted on Gescher but nodes are run with Linux 2.4.17-SMP

| **Processors** | **1N, 1P** | **1N, 4P** | **2N, 4P** | **3N, 4P** | **4N,4P** | **5N,4P** | **6N,4P** |
|---|---|---|---|---|---|---|---|
| Inspector | 0.089 | 0.022 | 0.0116 | 0.00798 | 0.00643 | 0.00489 | 0.00415 |
| Work distribution | 0.000258 | 0.000285 | 0.000318 | 0.000161 | 0.000204 | 0.01059 | 0.000142 |
| Update HALO | 0.149 | 3.114 | 9.110 | 16.170 | 24.060 | 33.868 | 43.830 |
| MPI_Init | 0.005 | 3.462 | 8.113 | 12.784 | 17.420 | 22.568 | 26.860 |
| Other | 0 | 0.001 | 0.007 | 0.016 | 0.010 | 0.004 | 0.036 |

**Table 7.3.** Control of parallelism overheads (s) for the HPF+ version for the backward pricing application.

| **Processors** | **1N, 1P** | **1N,4P** | **2N, 4P** | **3N, 4P** | **4N, 4P** | **5N,4P** | **6N,4P** |
|---|---|---|---|---|---|---|---|
| $T_{odata}$ | 0 | 0 | 0.01352 | 0.01432 | 0.01618 | 0.01753 | 0.01938 |
| $T_{octrl}$ | 0.1914 | 0.2457 | 1.8481 | 3.5198 | 4.8865 | 6.5339 | 7.9698 |
| $T_{olopa}$ | 0 | 2.413 | 1.535 | 1.215 | 1.055 | 0.9432 | 0.9175 |
| $T_{osyn}$ | 0.00401 | 0.7355 | 0.2878 | 0.224 | 0.1379 | 0.217 | 0.1025 |
| $T_i$ | 0.19541 | 3.39422 | 3.68448 | 4.97318 | 6.09565 | 7.71163 | 9.0091 |
| $T_u$ | 3.87599 | 1.30753 | 0.87239 | 0.65773 | 0.57228 | 0.48251 | 0.51576 |
| $To$ | 4.071 | 3.98175 | 4.556875 | 5.63091 | 6.66793 | 8.19415 | 9.525958 |
| $T_p$ | 320.488 | 83.086 | 44.109 | 31.999 | 26.444 | 24.015 | 22.71 |

**Table 7.4.** Overheads (s) of the OpenMP/MPI version for the backward pricing application. $T_{odata}, T_{olopa}, T_{octrl}$ and $T_{osyn}$ are data movement, loss of parallelism, control of parallelism, and synchronization overhead, respectively. $T_i$, $T_u$, $T_o$ are identified, unidentified and total overhead, respectively. $T_p$ is total execution time.

| **Processors** | **1N, 1P** | **1N,4P** | **2N, 4P** | **3N, 4P** | **4N, 4P** | **5N,4P** | **6N,4P** |
|---|---|---|---|---|---|---|---|
| Inspector | 0.00895 | 0.00891 | 0.051 | 0.0303 | 0.0228 | 0.018 | 0.0157 |
| Work distribution | 0.00614 | 0.000245 | 0.00599 | 0.00598 | 0.00594 | 0.000122 | 0.00593 |
| Update HALO | 0.142 | 0.140 | 0.594 | 1.122 | 1.349 | 1.783 | 2.136 |
| MPI_Init | 0.02874 | 0.005207 | 1.185 | 2.35 | 3.497 | 4.721 | 5.799 |
| Fork/join | 0.00559 | 0.0107 | 0.0116 | 0.0113 | 0.0104 | 0.0115 | 0.0129 |
| Other | 0.000005 | 0.000008 | 0.000226 | 0.000217 | 0.000234 | 0.000253 | 0.000249 |

**Table 7.5.** Control of parallelism overheads of the OpenMP/MPI version for the backward pricing application.

patched with `perfctr` for hardware counters measurement. We use MPICH 1.2.3 and `pgf90` compiler version 3.3 from the Portland Group Inc. The problem size is set to 200x200 points.

Load imbalance is applied for a given performance metric such as wall-clock time, cache accesses, etc., and analyzed in three modes: within a process which has multiple threads, within a node which has several processes, and among nodes. Figure 7.4 presents an example of load imbalance analysis of an `OMP DO` code region in subroutine `DO_JACOBI` (region numbered 28). This code region is not well load balance as wall-clock times spent in thread number 0 and 1 are significantly larger than that in thread 2 and 3.

Figure 7.5 displays the execution time summary for a single experiment with 4 SMP nodes and 1 process per node. For each SMP node a break-down of time spent in executing MPI statements, OpenMP parallel regions, and the

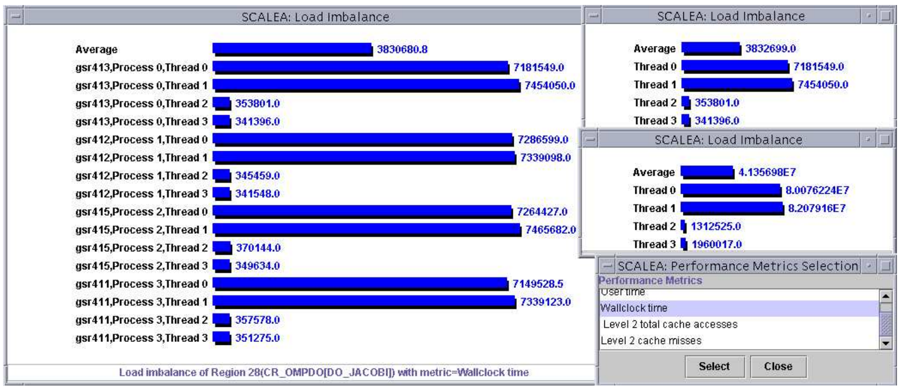

SCALEA: Load Imbalance
Average 3830680.8
gsr413,Process 0,Thread 0 7181549.0
gsr413,Process 0,Thread 1 7454050.0
gsr413,Process 0,Thread 2 353801.0
gsr413,Process 0,Thread 3 341396.0
gsr412,Process 1,Thread 0 7286599.0
gsr412,Process 1,Thread 1 7339098.0
gsr412,Process 1,Thread 2 345459.0
gsr412,Process 1,Thread 3 341548.0
gsr415,Process 2,Thread 0 7264427.0
gsr415,Process 2,Thread 1 7465682.0
gsr415,Process 2,Thread 2 370144.0
gsr415,Process 2,Thread 3 349634.0
gsr411,Process 3,Thread 0 7149528.5
gsr411,Process 3,Thread 1 7339123.0
gsr411,Process 3,Thread 2 357578.0
gsr411,Process 3,Thread 3 351275.0
Load imbalance of Region 28(CR_OMPDO[DO_JACOBI]) with metric=Wallclock time
SCALEA: Load Imbalance
Average 3832699.0
Thread 0 7181549.0
Thread 1 7454050.0
Thread 2 353801.0
Thread 3 341396.0
SCALEA: Load Imbalance
Average 4.135698E7
Thread 0 8.0076224E7
Thread 1 8.207916E7
Thread 2 1312525.0
Thread 3 1960017.0
SCALEA: Performance Metrics Selection
Performance Metrics
User time
Wallclock time
Level 2 total cache accesses
Level 2 cache misses
Select
Close

**Fig. 7.4.** Load imbalance analysis of an `OMP DO` code region in the subroutine `DO_JACOBI` executed with 4SMP nodes.

rest regions, is shown. A mouse click to an SMP-pie will make SCALEA to visualize detailed summary for processes in the SMP-pie. From this diagram it can be easily observed that the Stommel application is very communication intensive.

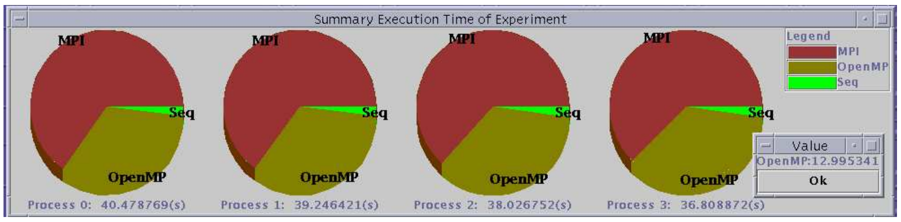
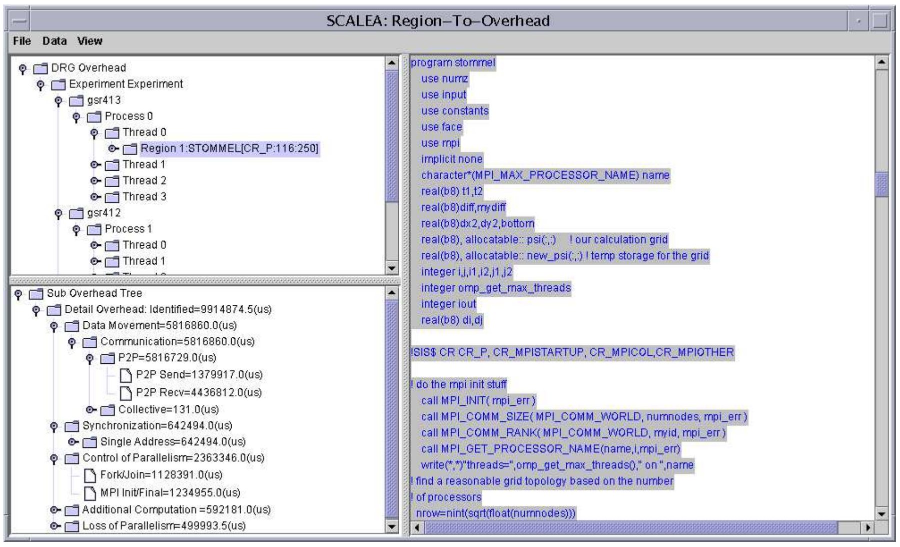


**Fig. 7.5.** Execution summary of Stommel executed with 4 SMP nodes.

SCALEA supports the programmer in the effort to examine detailed performance overheads for an experiment of a given program. Two modes are provided for this analysis. First, *Region-to-Overhead* mode (see Fig. 7.6) allows the programmer to select any code region instance in the DRG for which all detected performance overheads are displayed. Second, the *Overhead-to-Region* mode (see Fig. 7.7) enables the programmer to select the performance overhead of interest, based on which SCALEA displays the corresponding code region(s) in which the selected overhead occurs. This selection can be limited to a specific code region instance, thread or process. For both modi the source code of a region is only shown if the code region instance is selected in the DRG by a mouse click.

In Fig. 7.8 presents the execution time of Stommel in 6 experiments. Overall, Stomel does not scale well. The reason is mostly due to the high overhead, especially the communication, as presented in Figure 7.9.



```
DRG Overhead
  Experiment Experiment
    gsr413
      Process 0
        Thread 0
          Region 1:STOMMEL[CR_P:116:250]
        Thread 1
        Thread 2
        Thread 3
    gsr412
      Process 1
        Thread 0
        Thread 1
```

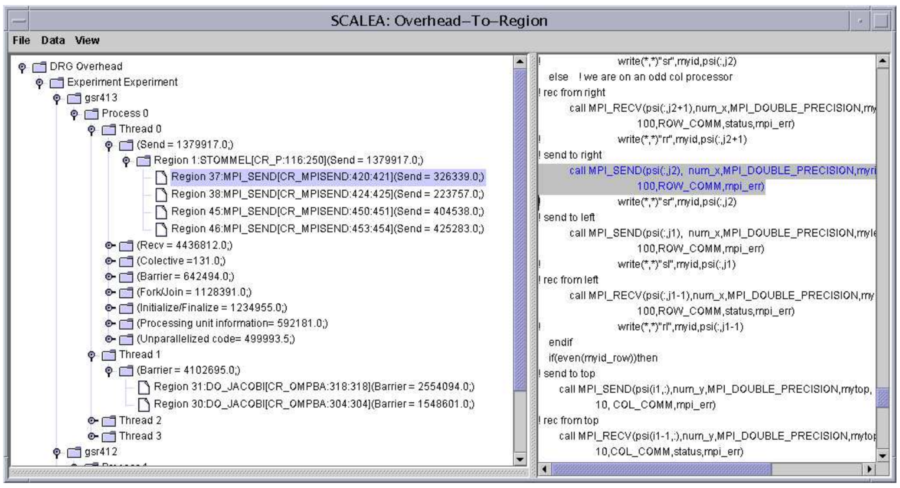
```
Sub Overhead Tree
  Detail Overhead: Identified=9914874.5(us)
    Data Movement=5816860.0(us)
      Communication=5816860.0(us)
        P2P=5816729.0(us)
          P2P Send=1379917.0(us)
          P2P Recv=4436812.0(us)
        Collective=131.0(us)
    Synchronization=642494.0(us)
      Single Address=642494.0(us)
    Control of Parallelism=2363346.0(us)
      Fork/Join=1128391.0(us)
      MPI Init/Final=1234955.0(us)
    Additional Computation =592181.0(us)
    Loss of Parallelism=499993.5(us)
```

```
program stommel
  use numz
  use input
  use constants
  use face
  use mpi
  implicit none
  character*(MPI_MAX_PROCESSOR_NAME) name
  real(b8) t1,t2
  real(b8)diff,mydiff
  real(b8)dx2,dy2,bottom
  real(b8), allocatable:: psi(:,:)    ! our calculation grid
  real(b8), allocatable:: new_psi(:,:) ! temp storage for the grid
  integer i,j,i1,i2,j1,j2
  integer omp_get_max_threads
  integer iout
  real(b8) di,dj

!SIS$ CR CR_P, CR_MPISTARTUP, CR_MPICOL,CR_MPIOTHER

! do the mpi init stuff
  call MPI_INIT( mpi_err )
  call MPI_COMM_SIZE( MPI_COMM_WORLD, numnodes, mpi_err )
  call MPI_COMM_RANK( MPI_COMM_WORLD, myid, mpi_err )
  call MPI_GET_PROCESSOR_NAME(name,i,mpi_err)
  write(*,*)"threads=",omp_get_max_threads()," on ",name
! find a reasonable grid topology based on the number
! of processors
 nrow=nint(sqrt(float(numnodes)))
```

**Fig. 7.6.** Region to Overhead mode for Stommel executed on 4 SMP nodes.



```
DRG Overhead
  Experiment Experiment
    gsr413
      Process 0
        Thread 0
          (Send = 1379917.0;)
            Region 1:STOMMEL[CR_P:116:250](Send = 1379917.0;)
              Region 37:MPI_SEND[CR_MPISEND:420:421](Send = 326339.0;)
              Region 38:MPI_SEND[CR_MPISEND:424:425](Send = 223757.0;)
              Region 45:MPI_SEND[CR_MPISEND:450:451](Send = 404538.0;)
              Region 46:MPI_SEND[CR_MPISEND:453:454](Send = 425283.0;)
          (Recv = 4436812.0;)
          (Colective =131.0;)
          (Barrier = 642494.0;)
          (Fork/Join = 1128391.0;)
          (Initialize/Finalize = 1234955.0;)
          (Processing unit information= 592181.0;)
          (Unparallelized code= 499993.5;)
        Thread 1
          (Barrier = 4102695.0;)
            Region 31:DO_JACOBI[CR_OMPBA:318:318](Barrier = 2554094.0;)
            Region 30:DO_JACOBI[CR_OMPBA:304:304](Barrier = 1548601.0;)
        Thread 2
        Thread 3
    gsr412
```

```
!                  write(*,*)"sr",myid,psi(:,j2)
  else   ! we are on an odd col processor
! rec from right
      call MPI_RECV(psi(:,j2+1),num_x,MPI_DOUBLE_PRECISION,my
                      100,ROW_COMM,status,mpi_err)
!                  write(*,*)"rr",myid,psi(:,j2+1)
! send to right
      call MPI_SEND(psi(:,j2),  num_x,MPI_DOUBLE_PRECISION,myri
                      100,ROW_COMM,mpi_err)
!                  write(*,*)"sr",myid,psi(:,j2)
! send to left
      call MPI_SEND(psi(:,j1),  num_x,MPI_DOUBLE_PRECISION,myle
                      100,ROW_COMM,mpi_err)
!                  write(*,*)"sl",myid,psi(:,j1)
! rec from left
      call MPI_RECV(psi(:,j1-1),num_x,MPI_DOUBLE_PRECISION,my
                      100,ROW_COMM,status,mpi_err)
!                  write(*,*)"rl",myid,psi(:,j1-1)
  endif
  if(even(myid_row))then
! send to top
     call MPI_SEND(psi(i1,:),num_y,MPI_DOUBLE_PRECISION,mytop,
              10, COL_COMM,mpi_err)
! rec from top
     call MPI_RECV(psi(i1-1,:),num_y,MPI_DOUBLE_PRECISION,mytop
              10,COL_COMM,status,mpi_err)
```

**Fig. 7.7.** Overhead to Region mode for Stommel executed on 4 SMP nodes.

### 7.3.4 3D Particle-In-Cell (3DPIC)

The 3D Particle-In-Cell application [102] simulates the interaction of high intensity ultrashort laser pulses with plasma in three dimensional geometry. This application (3DPIC) has been implemented as a Fortran90/MPI code.

We conducted several experiments by varying the machine size and by selecting the MPICH (version 1.2.3) communication library for Fast-Ethernet 100Mbps. The problem size (3D geometry) has been fixed with 30 cells in x-

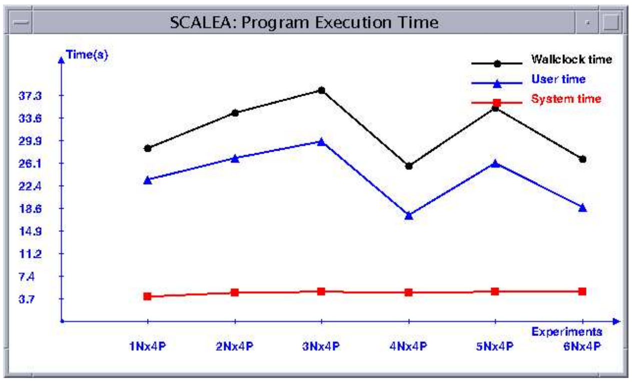
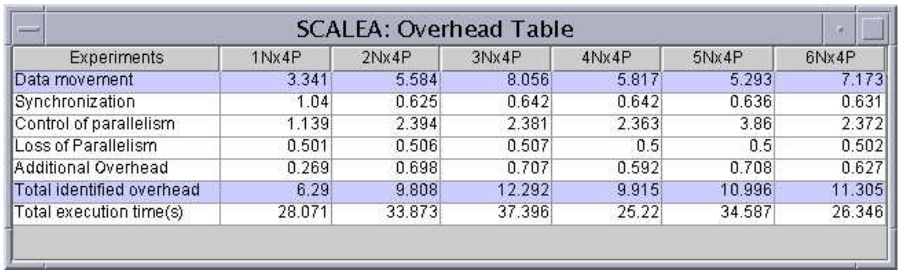


**Fig. 7.8.** Execution time of Stommel in 6 experiments. `1Nx4P` means 1 SMP node with 4 processors.



SCALEA: Overhead Table

| Experiments | 1Nx4P | 2Nx4P | 3Nx4P | 4Nx4P | 5Nx4P | 6Nx4P |
|---|---|---|---|---|---|---|
| Data movement | 3.341 | 5.584 | 8.056 | 5.817 | 5.293 | 7.173 |
| Synchronization | 1.04 | 0.625 | 0.642 | 0.642 | 0.636 | 0.631 |
| Control of parallelism | 1.139 | 2.394 | 2.381 | 2.363 | 3.86 | 2.372 |
| Loss of Parallelism | 0.501 | 0.506 | 0.507 | 0.5 | 0.5 | 0.502 |
| Additional Overhead | 0.269 | 0.698 | 0.707 | 0.592 | 0.708 | 0.627 |
| Total identified overhead | 6.29 | 9.808 | 12.292 | 9.915 | 10.996 | 11.305 |
| Total execution time(s) | 28.071 | 33.873 | 37.396 | 25.22 | 34.587 | 26.346 |

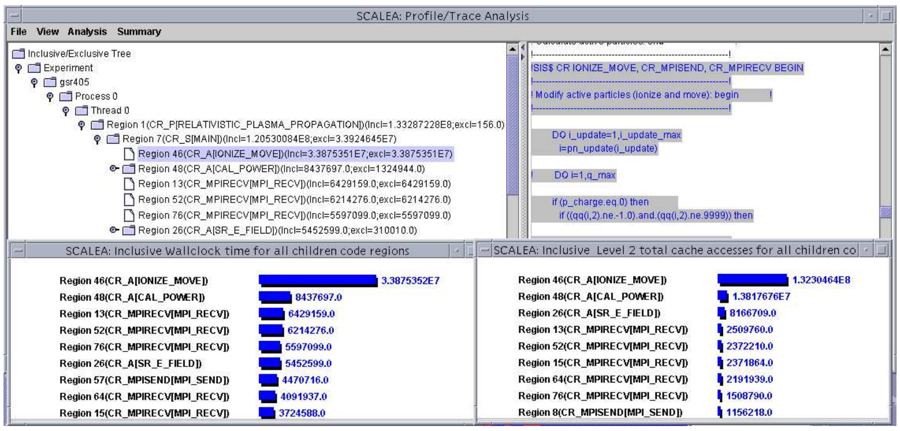
**Fig. 7.9.** Performance overheads of Stommel in 6 experiments.

direction (*nnx_glob*=30), 30 cells in y-direction (*nny_glob*=30), and 100 cells in z-direction (*nnz_glob*=100 ). The simulation has been done for 800 time steps (*itmax*=800).

### 7.3.4.1 Single Experiment Analysis Mode

SCALEA provides several analyses (e.g. *Load Imbalance Analysis, Inclusive/Exclusive Analysis, Metric Ratio Analysis, Overhead Analysis, Summary Analysis*) and diagrams to support performance evaluation based on a single execution of a program. In the following we just highlight some interesting results for a single experiment of the 3DPIC code with 3 SMP nodes and 4 CPUs per node.

The **Inclusive/Exclusive Analysis** is used to determine the execution time or overhead intensive code regions. The two lower-windows in Fig. 7.10 present the inclusive wall-clock times and the number of L2 cache accesses for sub-regions of the subroutine `MAIN` executed by thread 0 in process 0 of SMP node gsr405. The most time consuming region is `IONIZE_MOVE` because it is the most computation intensive region in 3DPIC which modifies the position of the particles by solving the equation of motion $\frac{d}{dt}(m\mathbf{v}) = q(\mathbf{E} + \frac{\mathbf{v}}{c} \times \mathbf{B})$ with a forth order Runge-Kutta routine. The related source code of region `IONIZE_MOVE` is shown in the upper-right window.



```
Inclusive/Exclusive Tree
  Experiment
    gsr405
      Process 0
        Thread 0
          Region 1(CR_P[RELATIVISTIC_PLASMA_PROPAGATION])(Incl=1.33287228E8;excl=156.0)
            Region 7(CR_S[MAIN])(Incl=1.20530084E8;excl=3.3924645E7)
              Region 46(CR_A[IONIZE_MOVE])(Incl=3.3875351E7;excl=3.3875351E7)
              Region 48(CR_A[CAL_POWER])(Incl=8437697.0;excl=1324944.0)
              Region 13(CR_MPIRECV[MPI_RECV])(Incl=6429159.0;excl=6429159.0)
              Region 52(CR_MPIRECV[MPI_RECV])(Incl=6214276.0;excl=6214276.0)
              Region 76(CR_MPIRECV[MPI_RECV])(Incl=5597099.0;excl=5597099.0)
              Region 26(CR_A[SR_E_FIELD])(Incl=5452599.0;excl=310010.0)
```

```
!-------------------------------------------------------!
!SIS$ CR IONIZE_MOVE, CR_MPISEND, CR_MPIRECV BEGIN
!-------------------------------------------------------!
! Modify active particles (ionize and move): begin      !
!-------------------------------------------------------!

      DO i_update=1,i_update_max
        i=pn_update(i_update)

!       DO i=1,q_max

      if (p_charge.eq.0) then
        if ((qq(i,2).ne.-1.0).and.(qq(i,2).ne.9999)) then
```

SCALEA: Inclusive Wallclock time for all children code regions

```
Region 46(CR_A[IONIZE_MOVE])      3.3875352E7
Region 48(CR_A[CAL_POWER])        8437697.0
Region 13(CR_MPIRECV[MPI_RECV])   6429159.0
Region 52(CR_MPIRECV[MPI_RECV])   6214276.0
Region 76(CR_MPIRECV[MPI_RECV])   5597099.0
Region 26(CR_A[SR_E_FIELD])       5452599.0
Region 57(CR_MPISEND[MPI_SEND])   4470716.0
Region 64(CR_MPIRECV[MPI_RECV])   4091937.0
Region 15(CR_MPIRECV[MPI_RECV])   3724588.0
```

SCALEA: Inclusive Level 2 total cache accesses for all children co

```
Region 46(CR_A[IONIZE_MOVE])      1.3230464E8
Region 48(CR_A[CAL_POWER])        1.3817676E7
Region 26(CR_A[SR_E_FIELD])       8166709.0
Region 13(CR_MPIRECV[MPI_RECV])   2509760.0
Region 52(CR_MPIRECV[MPI_RECV])   2372210.0
Region 15(CR_MPIRECV[MPI_RECV])   2371864.0
Region 64(CR_MPIRECV[MPI_RECV])   2191939.0
Region 76(CR_MPIRECV[MPI_RECV])   1508790.0
Region 8(CR_MPISEND[MPI_SEND])    1156218.0
```

**Fig. 7.10.** Inclusive/Exclusive analysis for sub-regions of the `MAIN` program.

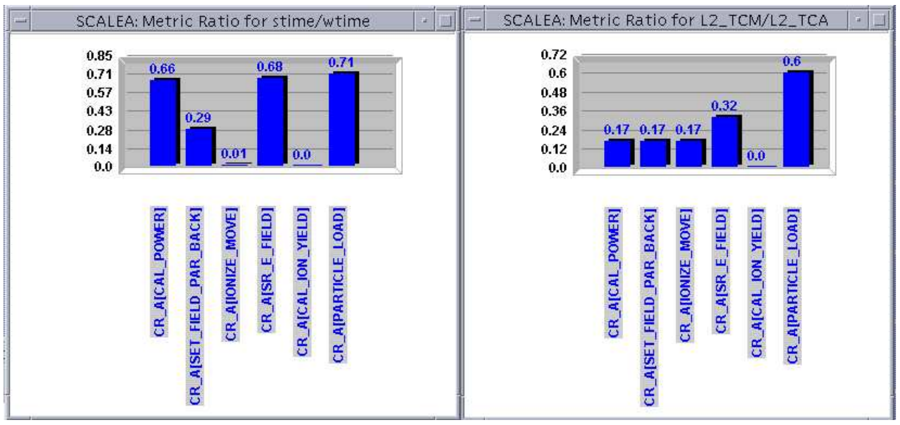
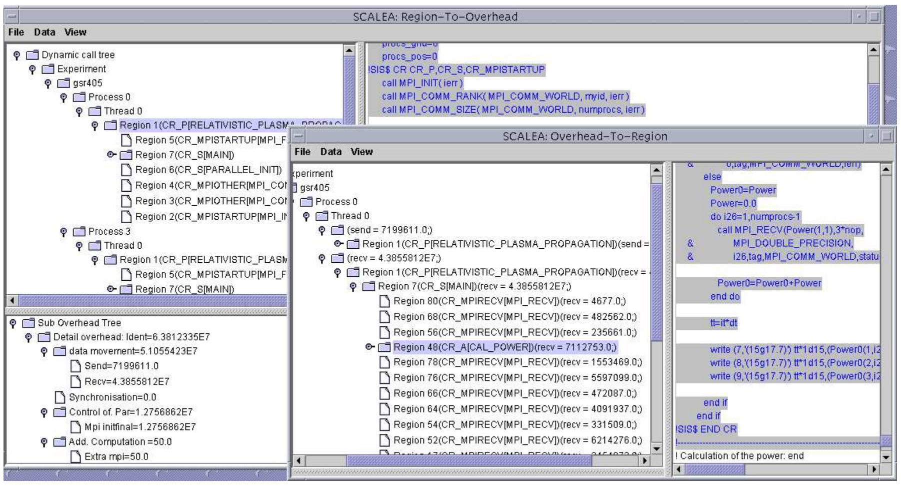


**Fig. 7.11.** Metric ratios for important code regions.

The **Metric Ratio Analysis** of SCALEA is then used to examine various important metric ratios (e.g., cache miss ratio, system time/wall-clock time, MFLOPS) of code regions in one experiment. Figure 7.11 shows the most critical system time/wall-clock time and L2 cache misses/L2 cache accesses ratios together with the corresponding code regions. The code regions `CAL_POWER`,`SET_FIELD_PAR_BACK`, `SR_E_FIELD`, and `PARTICLE_LOAD` imply a high system time/wall-clock time ratio due to expensive MPI constructs (included in system time). Both ratios are rather low for region `IONIZE_MOVE` because this region represents the computational part without any communication (mostly user time). The code region `PARTICLE_LOAD` shows a very high L2 cache misses/L2 cache accesses ratio because it initializes all particles in the 3D volume without accessing it again (little cache reuse).

The **Overhead Analysis** is used to investigate performance overheads for an experiment based on the overhead classification. In Fig. 7.12, SCALEA's *Region-To-Overhead* analysis (see the left-most window) examines the overheads of the code region instance numbered 1 (thread 0, process 0, node gsr405) which corresponds to the main program. The main program is dominated by the data movement overhead (see the lower-left window) which can be further refined to the overhead associated with send and receive operations. We then use the *Overhead-To-Region analysis* to inspect regions that cause receive overhead for thread 0, process 0, and node gsr405 (see the Overhead-To-Region window in Fig. 7.12). The largest portion of the receive overhead in subroutine `MAIN` is found in region `CAL_POWER` (7.11 seconds out of 43.85 seconds execution time).



```
Dynamic call tree
  Experiment
    gsr405
      Process 0
        Thread 0
          Region 1(CR_P[RELATIVISTIC_PLASMA_PROPAG
            Region 5(CR_MPISTARTUP[MPI_F
            Region 7(CR_S[MAIN])
            Region 6(CR_S[PARALLEL_INIT])
            Region 4(CR_MPIOTHER[MPI_CO
            Region 3(CR_MPIOTHER[MPI_CO
            Region 2(CR_MPISTARTUP[MPI_I
      Process 3
        Thread 0
          Region 1(CR_P[RELATIVISTIC_PLASM
            Region 5(CR_MPISTARTUP[MPI_F
            Region 7(CR_S[MAIN])
```

```
Sub Overhead Tree
  Detail overhead: Ident=6.3812335E7
    data movement=5.1055423E7
      Send=7199611.0
      Recv=4.3855812E7
    Synchronisation=0.0
    Control of. Par=1.2756862E7
      Mpi initfinal=1.2756862E7
    Add. Computation =50.0
      Extra mpi=50.0
```

```
    procs_pos=0
!SIS$ CR CR_P,CR_S,CR_MPISTARTUP
    call MPI_INIT( ierr )
    call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
    call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
```

```
xperiment
 gsr405
  Process 0
   Thread 0
    (send = 7199611.0;)
      Region 1(CR_P[RELATIVISTIC_PLASMA_PROPAGATION])(send =
    (recv = 4.3855812E7;)
      Region 1(CR_P[RELATIVISTIC_PLASMA_PROPAGATION])(recv =
        Region 7(CR_S[MAIN])(recv = 4.3855812E7;)
          Region 80(CR_MPIRECV[MPI_RECV])(recv = 4677.0;)
          Region 68(CR_MPIRECV[MPI_RECV])(recv = 482562.0;)
          Region 56(CR_MPIRECV[MPI_RECV])(recv = 235661.0;)
          Region 48(CR_A[CAL_POWER])(recv = 7112753.0;)
          Region 78(CR_MPIRECV[MPI_RECV])(recv = 1553469.0;)
          Region 76(CR_MPIRECV[MPI_RECV])(recv = 5597099.0;)
          Region 66(CR_MPIRECV[MPI_RECV])(recv = 472087.0;)
          Region 64(CR_MPIRECV[MPI_RECV])(recv = 4091937.0;)
          Region 54(CR_MPIRECV[MPI_RECV])(recv = 331509.0;)
          Region 52(CR_MPIRECV[MPI_RECV])(recv = 6214276.0;)
```

```
     else
       Power0=Power
       Power=0.0
       do i26=1,numprocs-1
         call MPI_RECV(Power(1,1),3*nop,
  &           MPI_DOUBLE_PRECISION,
  &           i26,tag,MPI_COMM_WORLD,statu

         Power0=Power0+Power
       end do

       tt=it*dt

       write (7,'(15g17.7)') tt*1d15,(Power0(1,i2
       write (8,'(15g17.7)') tt*1d15,(Power0(2,i2
       write (9,'(15g17.7)') tt*1d15,(Power0(3,i2

     end if
   end if
!SIS$ END CR
!---------------------------------------------
! Calculation of the power: end
```

**Fig. 7.12.** Overhead-To-Region and Region-To-Overhead analysis

An **Execution Summary Analysis** has been employed to examine the impact of communication on the execution time of the entire program. Figure 7.13 and 7.14 depict the *execution time summary* for the experiment executed with 3 SMP nodes and with 1 SMP node (4 processors per node), respectively. In the top window of Fig. 7.13 each pie represents one SMP node and each pie slice value corresponds to the average value across all processes of an SMP node (min/max/average values can be selected). By clicking onto an SMP pie, SCALEA displays a detailed summary for all processes in this node in the three lower windows of Fig. 7.13. Each pie is broken down into time spent in MPI routines and the remainder. Clearly, SCALEA indicates the dramatic increase of communication time when increasing number of SMP nodes. The MPI portion of the overall execution time corresponds approximately to 8.5% for the single S MP node version which raises to approximately 46% for 3 SMP-nodes. The experiments are based on a smaller problem size. Therefore, the communication to execution time ratio is rather high.

#### 7.3.4.2 Multiple Experiment Analysis Mode

SCALEA provides various features to support multi-experiment analysis that can be applied to single or multiple region(s). Figure 7.15 visualizes the execution time and speedup/improvement of the entire 3DPIC application, respectively. With increasing machine sizes, also the system time raises (close to the user time for 25 CPUs) due to the extensive communication.

Overall, 3DPIC doesn't scale well for the problem size considered because of the poor communication behavior (see Fig. 7.16) and also due to control of parallelism (for instance, initialization and finalization MPI operations).

### 7.3.5 LAPW0

LAPW0, which is part of the Wien2K package [46], is a material science program that calculates the effective potential of the Kohn-Sham eigen-value problem. LAPW0 has been implemented as a Fortran90 MPI code.

#### 7.3.5.1 First Series of Experiments

We first conduct experiments based on a single problem size and a fixed communication library and target machine network. Our experiments have been executed on Gescher with node is run with Linux 2.2.18-SMP patched with `perfctr` for hardware counters performance. We use MPICH 1.2.1 and `pgf90` compiler version 3.2 from the Portland Group Inc.

We use SCALEA to localize the most important code regions of LAPW0 which can be further subdivided into

- sequentialized code regions: `FFT_REAN0, FFT_REAN3, FFT_REAN4`
- parallelized code regions: `Interstitial_Potential, Loop_50`, `ENERGY, OUTPUT`

The execution time behavior and speedups (based on the sequential execution time of each code region) for each of these code regions are shown in Figures 7.17 and 7.18, respectively. LAPW0 has been examined for a problem size of 36 atoms which are distributed onto the processors of a set of SMP nodes. Clearly when using 8, 16, and 24 processors we can't reach optimal

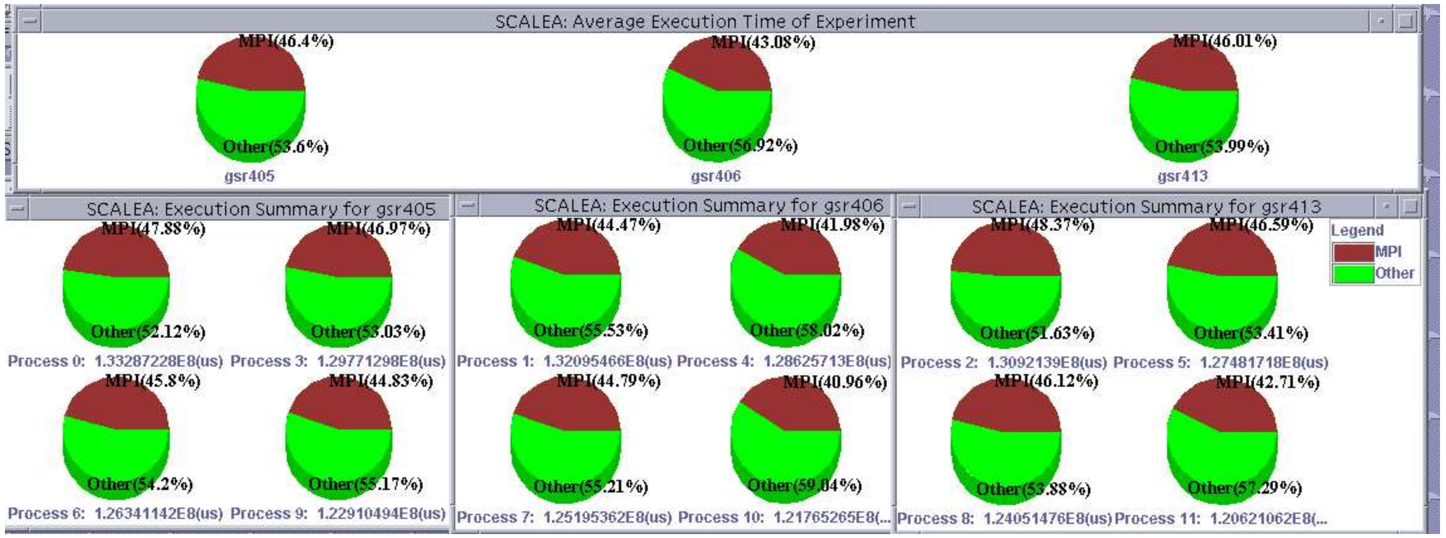


**Fig. 7.13.** Execution time summary for an experiment with 3 SMP-nodes

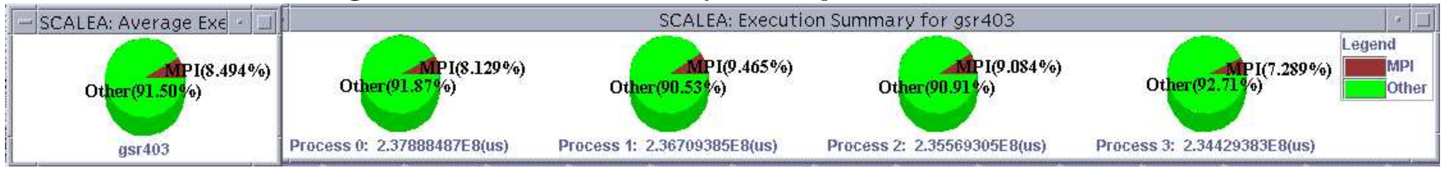


**Fig. 7.14.** Execution time summary for an experiment with 1 SMP-node

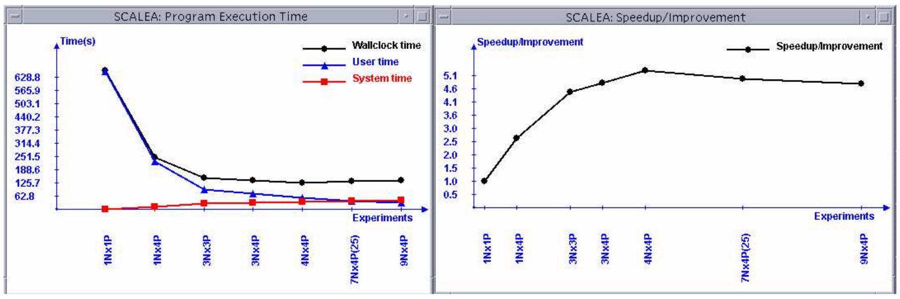
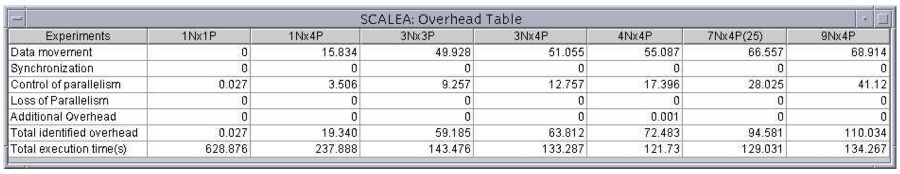


**Fig. 7.15.** Overall execution time and speedup/improvement for 3DPIC. `1Nx4P` means 1 SMP node with 4 processors (in case of 7Nx4P only 25 processors are used)


SCALEA: Overhead Table

| Experiments | 1Nx1P | 1Nx4P | 3Nx3P | 3Nx4P | 4Nx4P | 7Nx4P(25) | 9Nx4P |
|---|---|---|---|---|---|---|---|
| Data movement | 0 | 15.834 | 49.928 | 51.055 | 55.087 | 66.557 | 68.914 |
| Synchronization | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Control of parallelism | 0.027 | 3.506 | 9.257 | 12.757 | 17.396 | 28.025 | 41.12 |
| Loss of Parallelism | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Additional Overhead | 0 | 0 | 0 | 0 | 0.001 | 0 | 0 |
| Total identified overhead | 0.027 | 19.340 | 59.185 | 63.812 | 72.483 | 94.581 | 110.034 |
| Total execution time(s) | 628.876 | 237.888 | 143.476 | 133.287 | 121.73 | 129.031 | 134.267 |

**Fig. 7.16.** Performance overheads of the 3DPIC application. Note that total and unidentified overhead are missing as no sequential code version is available for the 3DPIC.

load balance, whereas 1, 2, 4, 6, 12 and 18 processors display a much better load imbalance. This effect is confirmed by SCALEA (see Figure 7.18) for the the most computationally intensive routines of LAPW0 (`Interstitial Potential` and `Loop_50`).

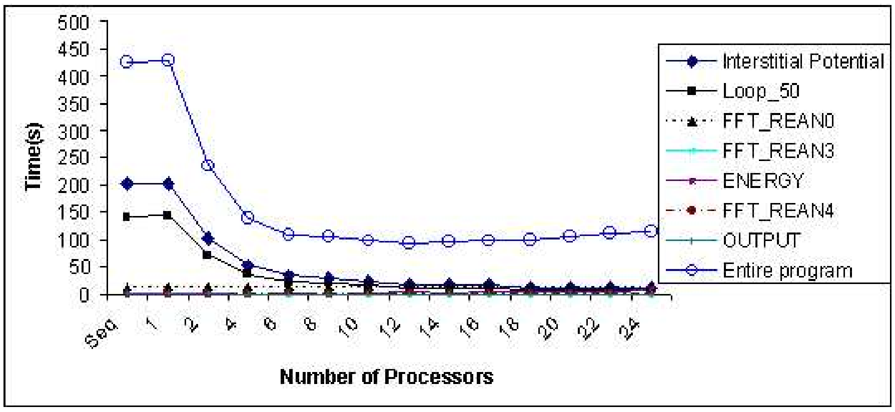


**Fig. 7.17.** Execution times for LAPW0 code regions.

| **Processors** | **Seq** | **1** | **4** | **8** | **12** | **16** | **20** | **24** |
|---|---|---|---|---|---|---|---|---|
| $T_{olopa}$ | | 0 | 12.047 | 14.222 | 14.83258 | 15.356 | 15.8289 | 15.9179 |
| $T_{odata}$ | | 0.000015 | 1.267 | 1.361 | 1.594 | 2.677 | 3.298 | 7.993 |
| $T_{octrp}$ | | 0.00248 | 6.58 | 12.99 | 18.15 | 21.39 | 26.62 | 31.56 |
| $T_i$ | | 0.002495 | 19.894 | 28.573 | 34.576 | 39.423 | 45.7469 | 55.4709 |
| $T_u$ | | 2.601505 | 13.004 | 23.043 | 23.388 | 31.328 | 41.811 | 41.633 |
| $T_o$ | | 2.604 | 32.898 | 51.617 | 57.965 | 70.751 | 87.558 | 97.104 |
| $T_p$ | 425.207 | 427.811 | 139.2 | 104.768 | 93.399 | 97.327 | 108.819 | 114.821 |

**Table 7.6.** Overheads (s) of LAWP0. $T_{odata}$, $T_{olopa}$ and $T_{octrl}$ are data movement, loss of parallelism, and control of parallelism overhead, respectively. $T_i$, $T_u$, $T_o$ are identified, unidentified and total overhead, respectively. $T_p$ is total execution time.

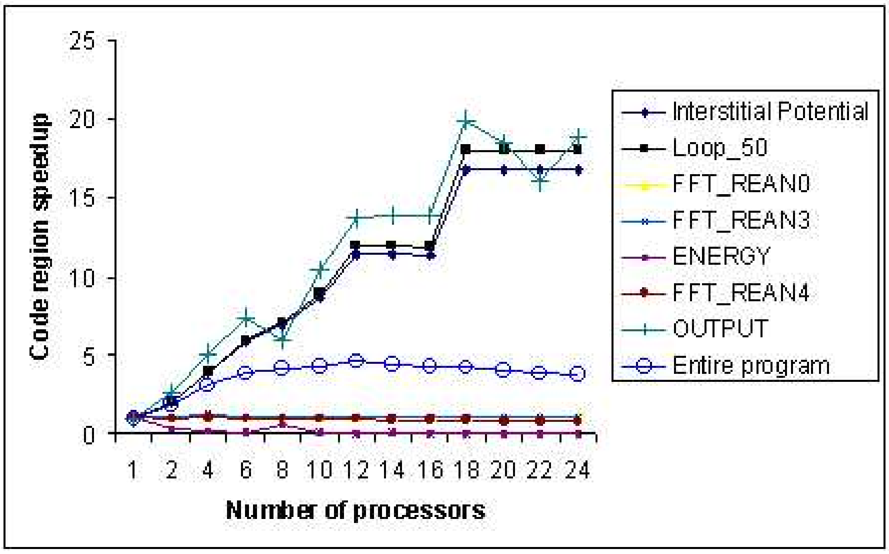


**Fig. 7.18.** Speedup values for LAPW0 code regions.

Overall, LAPW0 scales poorly due to load imbalances and large overheads due to loss of parallelism, data movement, and synchronization (see Table 7.6). LAPW0 uses many BLAS and SCALAPACK library calls that are currently not instrumented by SCALEA which is the reason for the large fraction of unidentified overhead (see Table 7.6). The main sources of the control of parallelism overhead is caused by `MPI_Init` (see Figure 7.19). SCALEA also discovered the main subroutines that cause the loss of parallelism overhead: `FFT_REAN0`, `FFT_REAN3`, and `FFT_REAN4` all of which are sequentialized.

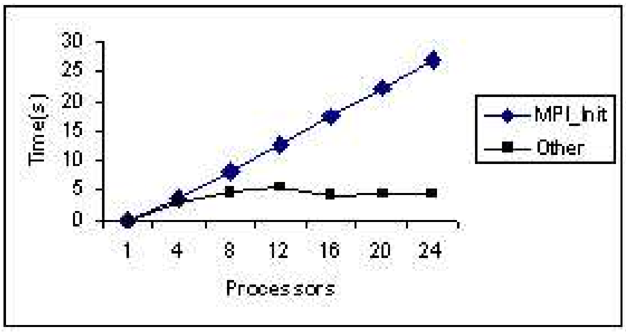


**Fig. 7.19.** Sources of control of parallelism overhead for LAPW0.

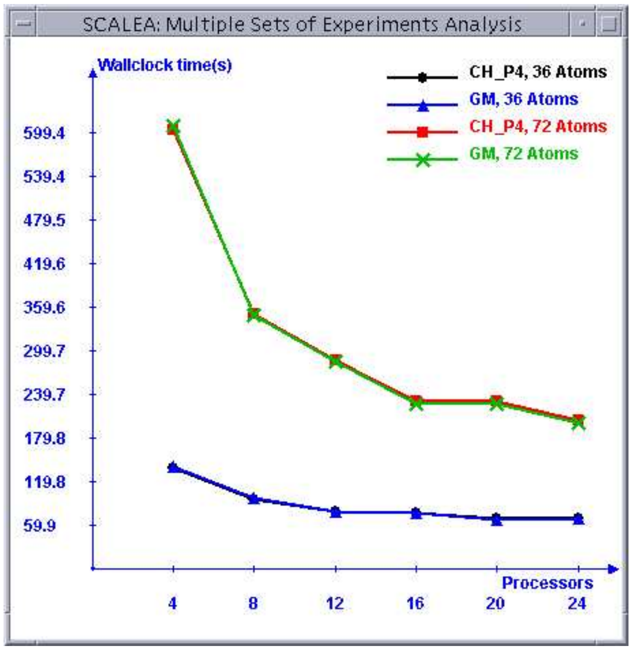


**Fig. 7.20.** Execution time of LAPW0 with 36 and 72 atoms. *CH_P4, GM* means that MPICH 1.2.3 has been used for CH_P4 (for Fast-Ethernet 100Mbps) and Myrinet, respectively.

### 7.3.5.2 Second Series of Experiments

In the first series of experiments, we conducted experiments with single problem size. Upon that analysis, both LAPW0 and the cluster has been updated. In this section, we outline some additional performance analyses for different machine and problem sizes (36 and 72 atoms) with two different networks. We refined the instrumentation and measurement of code regions `Interstitial`

`Potential` and `Loop_50`. The code region `Interstitial Potential` is divided into `CA_COULOMB_INTERSTITIAL_POTENTIAL` and `CA_MULTIPOLMENTS`. The code region `Loop_50` is divided into `CAL_CP_INSIDE_SPHERES` and `CAL_COULOMB _RMT`. SMP nodes are connected by Ethernet 100MB and Myrinet. Node is run with Linux 2.4 .17-SMP.

The overall execution times provided by SCALEA's **Multi-Set Experiment Analysis** are shown in Fig. 7.20. The first observation is that changing the communication library and communication network from MPICH CH_P4 to MPICH GM does not lead to a performance improvement of the parallel LAPWO version. The second observation is that we cannot achieve better performance by varying the number of processors from 12 to 16 and 20 to 24 processors for the experiments with 36 atoms, or by varying the processor number from 20 to 24 processors for the 72 atom experiment. In order to explain this behavior we concentrate on the experiment with 36 atoms. Figure 7.21 visualizes the execution times for the most computational intensive code regions of LAPW0 supported by **Multiple Region Analysis**. The execution times of these code regions remain almost constant although the number of processors is increased from 12 to 24 and from 20 to 24. The LAPW0 code exposes a load balancing problem for 16, 20, an d 24 processors. Moreover, code regions `FFT_REAN0, FFT_REAN3`, and `FFT_REAN4` are executed sequentially, as shown in Fig. 7.21. These code regions cause a large of loss of parallelism overhead (see Fig. 7.22).

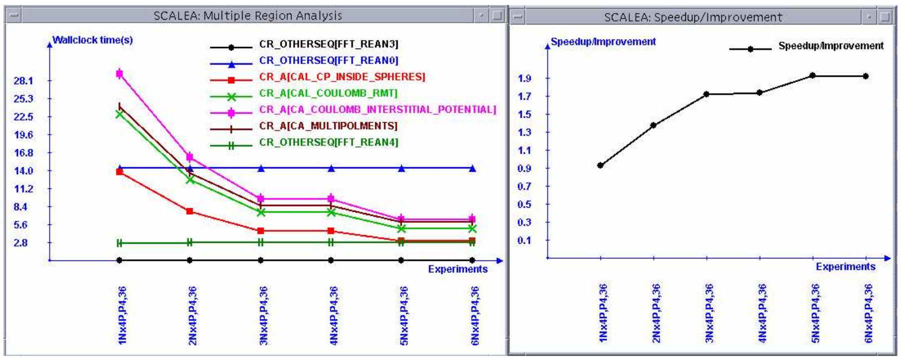
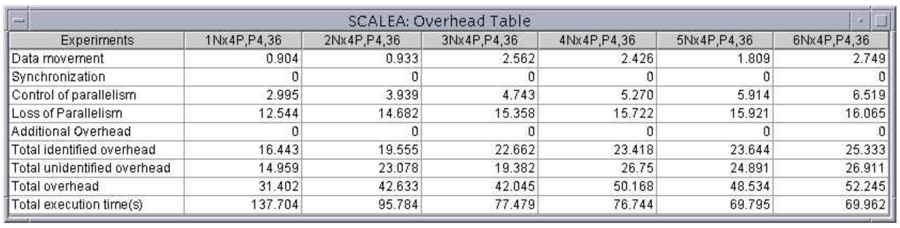


**Fig. 7.21.** Execution time of computationally intensive code regions. *1 Nx4P,P4,36* means 1 SMP node with 4 processors using MPICH CH_P4 and the problem size is 36 atoms.

In summary, LAPW0 scales poorly due to load imbalances and a large loss of parallelism overhead caused by the lack of parallelizing the `FFT_REAN0, FFT_REAN3`, and `FFT_REAN4` code regions. In order to improve the performance of this application, these code regions must be parallelized as well.



SCALEA: Overhead Table

| Experiments | 1Nx4P,P4,36 | 2Nx4P,P4,36 | 3Nx4P,P4,36 | 4Nx4P,P4,36 | 5Nx4P,P4,36 | 6Nx4P,P4,36 |
|---|---|---|---|---|---|---|
| Data movement | 0.904 | 0.933 | 2.562 | 2.426 | 1.809 | 2.749 |
| Synchronization | 0 | 0 | 0 | 0 | 0 | 0 |
| Control of parallelism | 2.995 | 3.939 | 4.743 | 5.270 | 5.914 | 6.519 |
| Loss of Parallelism | 12.544 | 14.682 | 15.358 | 15.722 | 15.921 | 16.065 |
| Additional Overhead | 0 | 0 | 0 | 0 | 0 | 0 |
| Total identified overhead | 16.443 | 19.555 | 22.662 | 23.418 | 23.644 | 25.333 |
| Total unidentified overhead | 14.959 | 23.078 | 19.382 | 26.75 | 24.891 | 26.911 |
| Total overhead | 31.402 | 42.633 | 42.045 | 50.168 | 48.534 | 52.245 |
| Total execution time(s) | 137.704 | 95.784 | 77.479 | 76.744 | 69.795 | 69.962 |

**Fig. 7.22.** Performance overheads for LAPW0.

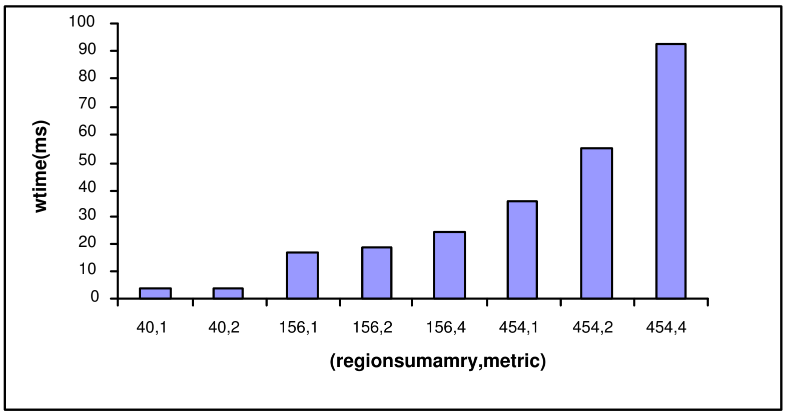


**Fig. 7.23.** Execution time for fuzzy C-means clustering with 4 clusters, algorithm terminating criteria $\epsilon = 0.01$, and degree of fuzziness of the clustering $m = 2$.

### 7.3.6 Fuzzy C-Mean Clustering Experiments

#### 7.3.6.1 Performance Analysis of the Fuzzy C-means Algorithm

Figure 7.23 displays execution times of the our fuzzy C-means clustering for some experiments. The clustering analysis is executed on a Sun Blade 150 (UltraSPARC-IIe 550MHz) workstation with 768MB. Time is measured by using Java `System.currentTimeMillis()` call. When the number of region summaries and the number of performance metrics increase, the execution time quickly increases. Therefore, with a large dataset of multiple performance metrics, we may need to filter data to be clustered.

### 7.3.7 Clustering Analysis of Performance Experiments

Figure 7.24 presents an example of fuzzy C-means clustering for LAPW0 with a single metric `wtime`. The resulting clusters reveal that there are few code regions that have high execution time when compared to the execution time of the whole application; these regions have high graded membership

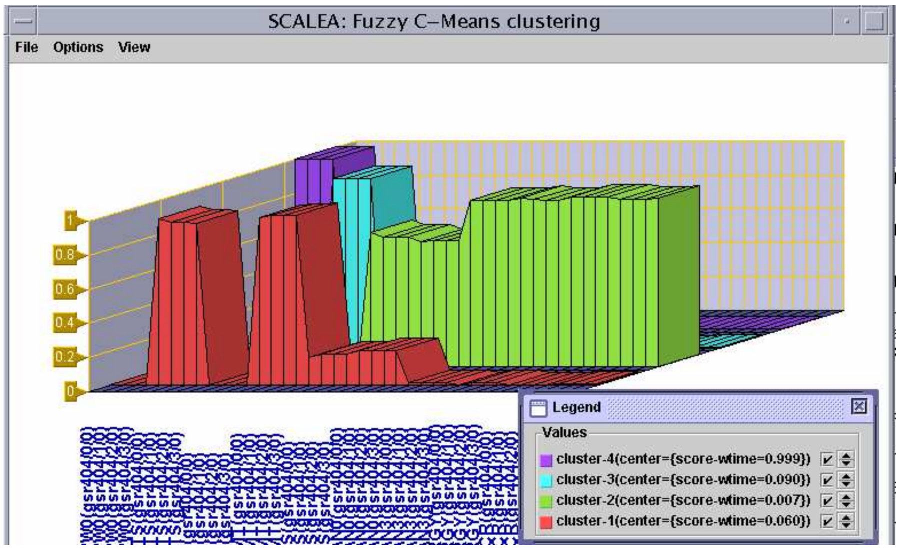

SCALEA: Fuzzy C–Means clustering
File Options View
1
0.8
0.6
0.4
0.2
0
Legend
Values
cluster-4(center={score-wtime=0.999})
cluster-3(center={score-wtime=0.090})
cluster-2(center={score-wtime=0.007})
cluster-1(center={score-wtime=0.060})

**Fig. 7.24.** Fuzzy C-means clustering (4 clusters, algorithm terminating criteria $\epsilon = 0.01$, degree of fuzziness of the clustering $m = 2$) for an experiment of LAPW0 (executed with 4 CPUs, MPICH CH_P4). Problem size is set to 36 atoms.

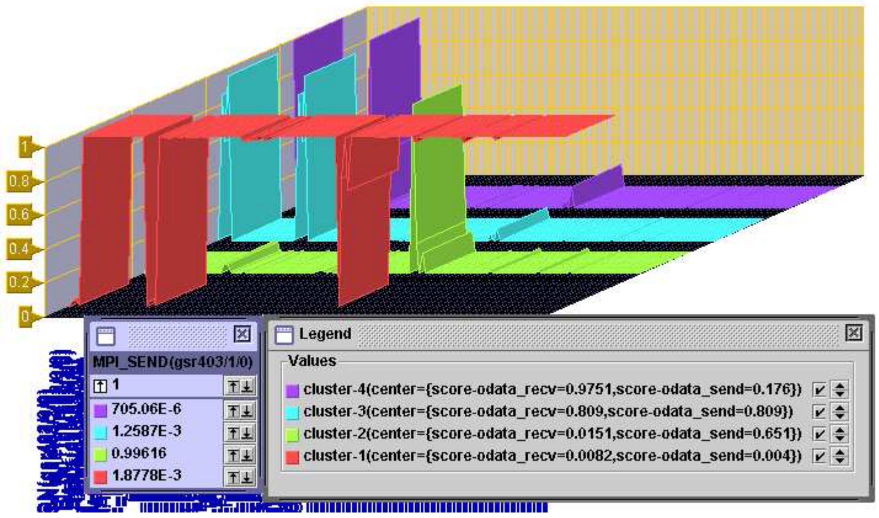


**Fig. 7.25.** Fuzzy C-means clustering (4 clusters, algorithm terminating criteria $\epsilon = 0.01$, degree of fuzziness of the clustering $m = 2$.) for an experiment of 3DPIC (executed with 4 CPUs, MPICH CH_P4).

in `cluster-4`. Most code regions measured have high membership degree in `cluster-1, cluster-2` and `cluster-3`. However, these three clusters have low centroid values, suggesting that the code regions having high graded membership in these clusters have little impact on the whole performance of the program.

Figure 7.25 displays the result of the fuzzy C-means clustering for instrumented code regions of 3DPIC with two performance metrics {`odata_recv, odata_send`}. Figure 7.25 shows that there are many code regions caused both send and receive overhead. The membership indicates the degree of overhead compared with the total overhead in a process of the experiment. Cluster `cluster-1` classifies many code regions which contribute very little send and receive overheads. Few code regions that have high send overhead but low receive overhead are in cluster `cluster-2`. Code regions with high graded membership in `cluster-3` indicate both high send and receive overheads. In `cluster-4`, just few code regions have very high receive overheads but low send overheads. Code regions in `cluster-3, cluster-4` are not `MPI_Send` calls and `MPI_Recv` but those contain `MPI_Send` and `MPI_Recv` calls. Most instrumented `MPI_Send` and `MPI_Recv` code regions contribute little overhead; these code regions are clustered into `cluster-1`. However many such code regions make total send and receive overhead high.

## 7.4 Performance Monitoring and Analysis for the Grid

In this section we present a few experiments of SCALEA-G in the Grid. Unless stated otherwise, experiments are conducted on the Grid test-bed mentioned below.

### 7.4.1 Grid Test-bed

We have deployed our sensor-based monitoring infrastructure on three domains: VCPC (University of Vienna), UIBK (University of Innsbruck) and GUP (Linz University) in the Austrian Grid [28]. Figure 7.26 presents our experimental test-bed. We set up three SM groups named SM-VCPC, SM-UIBK and SM-GUP in VCPC, UIBK, GUP, respectively. We establish a DS group that includes one DS in VCPC (DS-VCPC) and one in UIBK (DS-UIBK). Each DS stores data in a PostgreSQL database server which can be executed on the same domain (e.g. in case of DS-UIBK) or different one (e.g. DS-VCPC). SM stores collected data into XML containers implemented atop Berkeley DB XML [11]. There are two Registries in VCPC and UIBK. A client is deployed in PAR domain (in University of Vienna). All DSs and Registries can be accessed by all SMs and clients, but only SMs executed on `bridge/VCPC`, `olperer/UIBK`, `iris/GUP` can directly deliver data to the client executed on `kim/PAR` due to the firewall. Most machines in our test-bed are non dedicated.

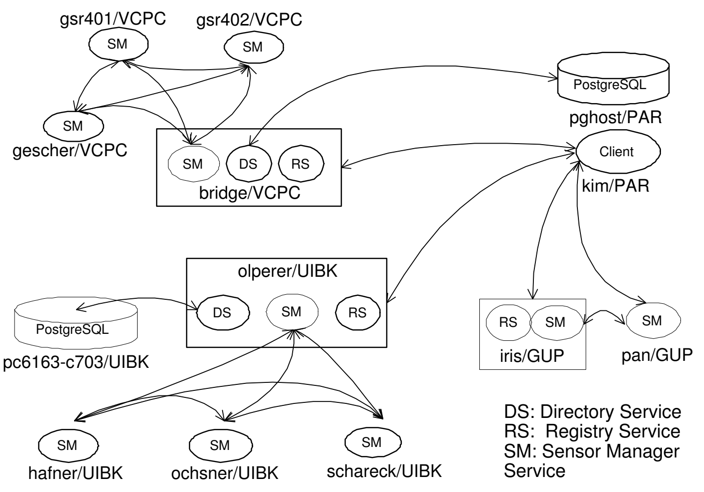


**Fig. 7.26.** Grid experimental test-bed.

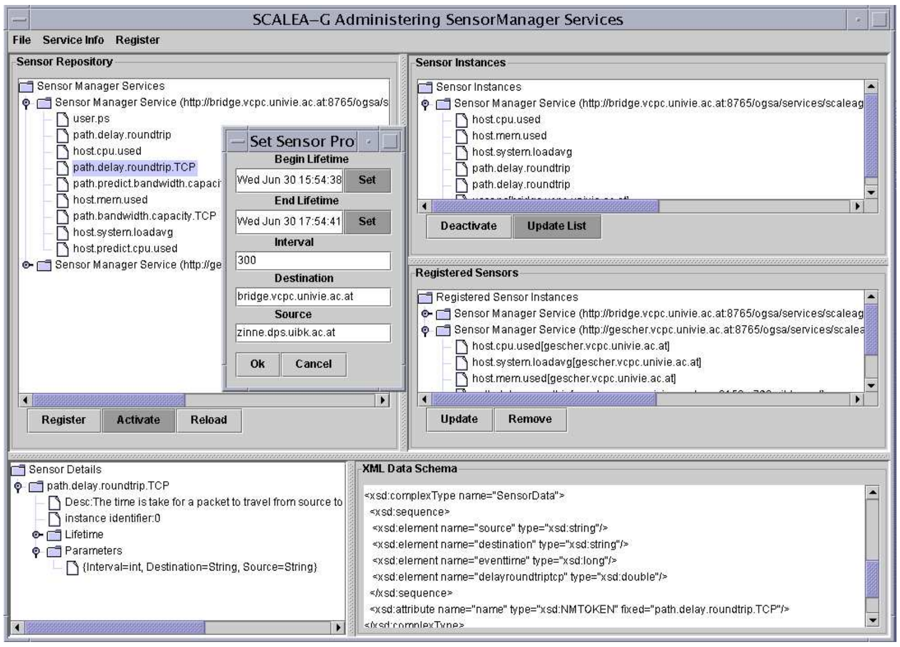
### 7.4.2 Administrating Sensor Manager Services and Sensors

Figure 7.27 presents the administration GUI which is used to manage activities of internal sensors of Sensor Manager Services. The administration GUI keeps a list of Sensor Manager Services to be managed. SCALEA-G administrator connects to selected Sensor Manager Services and receives a list of available sensors and a list of sensor instances (active sensors), as presented in the top-left and top-right window of Fig. 7.27, respectively. The administrator can make a request creating a new sensor instance by selecting a sensor and clicking the *Activate* button. Similarly, an existing sensor instance can be deactivated (by selecting *Deactivate* button). The administrator can control the registration (*Add, Update, Remove*) information about Sensor Manager Services, sensor instances and properties of data they provide to directory service.

The administrator can also activate or deactivate the data receiver in a Sensor Manager Service. If the data receiver is disable, the Sensor Manager Service will not receive data sent by sensor instances. This simple method allows the administrator to control the site/computational node on which data is collected. When activating the data receiver, the administrator can specify the port on which the data receiver is binding.

### 7.4.3 Performance Monitoring and Analysis User GUI

The user can monitor and analyze online performance behavior of Grid systems and applications at the same time. In the top-left window of Figure 7.28, the user can examine DSs and information about sensor instances registered



```
Sensor Manager Services
  Sensor Manager Service (http://bridge.vcpc.univie.ac.at:8765/ogsa/s
    user.ps
    path.delay.roundtrip
    host.cpu.used
    path.delay.roundtrip.TCP
    path.predict.bandwidth.capaci
    host.mem.used
    path.bandwidth.capacity.TCP
    host.system.loadavg
    host.predict.cpu.used
  Sensor Manager Service (http://ge
```

```
Begin Lifetime
Wed Jun 30 15:54:38
End Lifetime
Wed Jun 30 17:54:41
Interval
300
Destination
bridge.vcpc.univie.ac.at
Source
zinne.dps.uibk.ac.at
```

```
Sensor Instances
  Sensor Manager Service (http://bridge.vcpc.univie.ac.at:8765/ogsa/services/scaleag
    host.cpu.used
    host.mem.used
    host.system.loadavg
    path.delay.roundtrip
    path.delay.roundtrip
```

```
Registered Sensor Instances
  Sensor Manager Service (http://bridge.vcpc.univie.ac.at:8765/ogsa/services/scaleag
  Sensor Manager Service (http://gescher.vcpc.univie.ac.at:8765/ogsa/services/scalea
    host.cpu.used[gescher.vcpc.univie.ac.at]
    host.system.loadavg[gescher.vcpc.univie.ac.at]
    host.mem.used[gescher.vcpc.univie.ac.at]
```

```
Sensor Details
  path.delay.roundtrip.TCP
    Desc:The time is take for a packet to travel from source to
    instance identifier:0
    Lifetime
    Parameters
      {Interval=int, Destination=String, Source=String}
```

```
<xsd:complexType name="SensorData">
 <xsd:sequence>
  <xsd:element name="source" type="xsd:string"/>
  <xsd:element name="destination" type="xsd:string"/>
  <xsd:element name="eventtime" type="xsd:long"/>
  <xsd:element name="delayroundtriptcp" type="xsd:double"/>
 </xsd:sequence>
 <xsd:attribute name="name" type="xsd:NMTOKEN" fixed="path.delay.roundtrip.TCP"/>
</xsd:complexType>
```

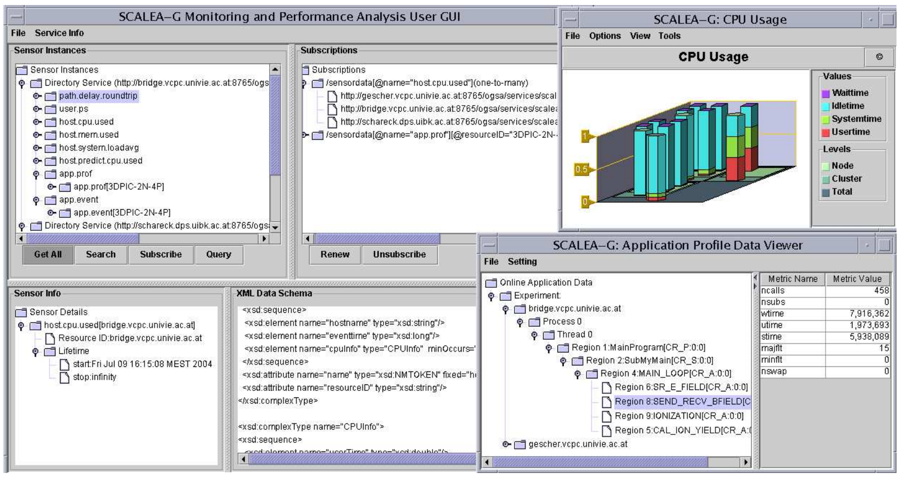
**Fig. 7.27.** SCALEA-G Administrator GUI

with these DSs. Under the DS tree, sensor instances are grouped into categories based on the sensor identifier; each category provides the same data type but of different resources. For each sensor instance, the user can examine its properties (e.g. lifetime and XML schema) by choosing the instance from the list of sensor instances.

The user can perform one-to-one or one-to-many DQS by selecting sensor instance (for one-to-one mode) or sensor category (for one-to-many mode), choosing `Subscribe` or `Query` (see Figure 7.28) and then editing the request (e.g. subscription time, XML data filter) if needed. We can subscribe, query and then examine multiple types of monitoring data at the same time. For example, we subscribed application data of experiment `3DPIC-2N-4P` and `host.cpu.used` data of computational nodes on which the application processes are executed, and then examined that data online (see *Application Profile Data Viewer* and *CPU Usage* window in Figure 7.28). A list of existing subscriptions is shown in the `Subscriptions` tree. A subscription can be canceled by selecting the subscription and clicking `Unsubscribe`. Similarly, an existing subscription can be renewed by clicking `Renew`.

## 7.4.4 Performance Analysis of Data Discovery

To evaluate the performance of the discovery of data providers within the test-bed, we setup two modes. In *one-to-one mode*, a client sends requests



```
Sensor Instances
  Directory Service (http://bridge.vcpc.univie.ac.at:8765/ogs
    path.delay.roundtrip
    user.ps
    host.cpu.used
    host.mem.used
    host.system.loadavg
    host.predict.cpu.used
    app.prof
      app.prof[3DPIC-2N-4P]
    app.event
      app.event[3DPIC-2N-4P]
  Directory Service (http://schareck.dps.uibk.ac.at:8765/ogs

Subscriptions
  /sensordata[@name="host.cpu.used"](one-to-many)
    http://gescher.vcpc.univie.ac.at:8765/ogsa/services/scal
    http://bridge.vcpc.univie.ac.at:8765/ogsa/services/scalea
    http://schareck.dps.uibk.ac.at:8765/ogsa/services/scalea
  /sensordata[@name="app.prof"][@resourceID="3DPIC-2N-

Sensor Details
  host.cpu.used[bridge.vcpc.univie.ac.at]
    Resource ID:bridge.vcpc.univie.ac.at
    Lifetime
      start:Fri Jul 09 16:15:08 MEST 2004
      stop:infinity
```

```
 <xsd:sequence>
  <xsd:element name="hostname" type="xsd:string"/>
  <xsd:element name="eventtime" type="xsd:long"/>
  <xsd:element name="cpuInfo" type="CPUInfo"  minOccurs='
 </xsd:sequence>
 <xsd:attribute name="name" type="xsd:NMTOKEN" fixed="h
 <xsd:attribute name="resourceID" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="CPUInfo">
<xsd:sequence>
```

```
Online Application Data
  Experiment:
    bridge.vcpc.univie.ac.at
      Process 0
        Thread 0
          Region 1:MainProgram[CR_P:0:0]
            Region 2:SubMyMain[CR_S:0:0]
              Region 4:MAIN_LOOP[CR_A:0:0]
                Region 6:SR_E_FIELD[CR_A:0:0]
                Region 8:SEND_RECV_BFIELD[C
                Region 9:IONIZATION[CR_A:0:0]
                Region 5:CAL_ION_YIELD[CR_A:
    gescher.vcpc.univie.ac.at
```

| Metric Name | Metric Value |
|---|---|
| ncalls | 458 |
| nsubs | 0 |
| wtime | 7,916,362 |
| utime | 1,973,693 |
| stime | 5,938,089 |
| majflt | 15 |
| minflt | 0 |
| nswap | 0 |

**Fig. 7.28.** Performance monitoring and analysis GUI

directly to DS which in turns finds data providers of the requested data. If the DS cannot locate the data provider, it will not send the request to other DSs in the same group. In *group mode*, if the DS cannot answer the request, it sends the request to its edge peers in its group asking for the location of data providers.

At a random time, for each mode we conduct a series of 20 tests with the interval 60 seconds between two consecutive tests. Each test is repeated 10 times and we select the best timing value as result of the test. In both modes, a client in PAR domain sends requests to DSs at the same time. We also measure the latency of *ping operation* from clients to DSs. The time is measured by using Java `System.currentTimeMillis()` call.

Figure 7.29 presents search time in one-to-one and group mode, and latency of `ping` operation. Overall for both DSs, the ping latency is larger than a half of the time spent on the search of data providers in one-to-one mode, suggesting that the time DS spends on searching its database is small when compared with ping latency. A considerable portion of time spent in the discovery process is service-operation latency from client to service. In our implementation, for group mode, DS creates a new thread which calls an edge peer when the DS cannot locate the data provider. Search time in group mode nearly doubles that in one-to-one mode partially because at the same time a DS has to fulfill a request from an edge DS and to forward a request to its edge DS. The latency from client domain (PAR) to DS-UIBK is higher than that to DS-VCPC, also DS-VCPC is executed on an SMP machine where DS-UIBK is executed a single CPU machine. Therefore, conducting group-based and one-to-one discovery through DS-VCPC is considerably faster than that via DS-UIBK due to the differences of network latency and computation

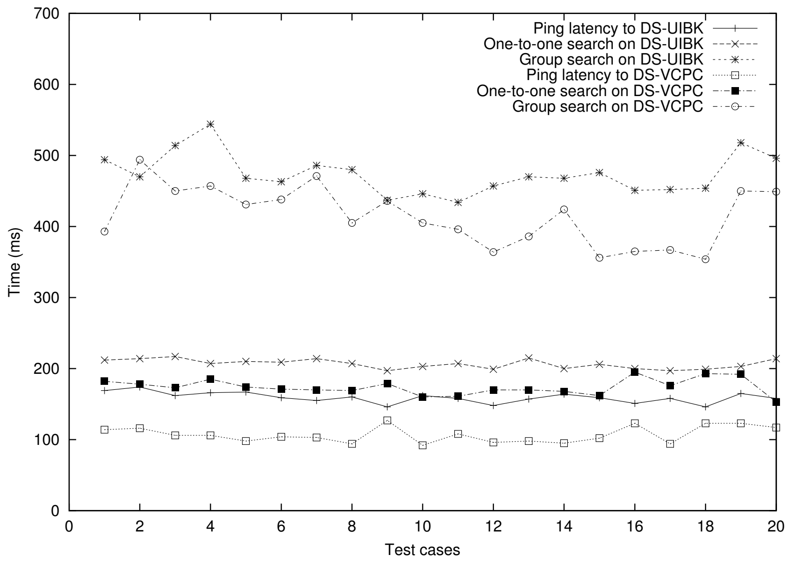


**Fig. 7.29.** Ping latency and search time.

power. Also search times in group mode are high variance partially because the search now involves different DSs running on different networks.

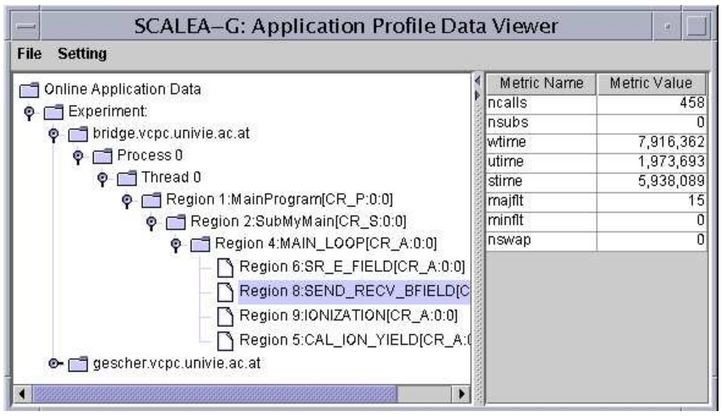
### 7.4.5 Monitoring and Performance Data Integration

We have implemented a GUI client which accesses a variety of types of monitoring data provided by the middleware and conducts the analysis of that data. We present some experiments of using that GUI client to monitor and analysis the monitoring data.

Figure 7.30 presents an analysis of profiling data collected by application sensors. Incremental profiling data of Grid applications is sent online to SMs. The application profile analyzer then conducts DQS on profiling data, analyzing and visualizing the result to the user. The left window of Figure 7.30 shows code regions associated with their processing units (computational node, process, thread). For each code region, performance metrics are incrementally updated in the right window.

Figure 7.31 presents an example of bandwidth monitoring of a network path from VCPC to UIBK. We setup a simple rule set based on fuzzy variable for the bandwidth, as presented in Section 6.2.3.7. Only when the bandwidth of the network path is *very low, low* and *very high*, the sensor sends events to SM. Events are subscribed and visualized by a simple generic event viewer, as shown in Figure 7.31.

Figure 7.32 presents a few snapshots of monitoring system load, CPU usage and network delay roundtrip (monitoring data provided by event-driven



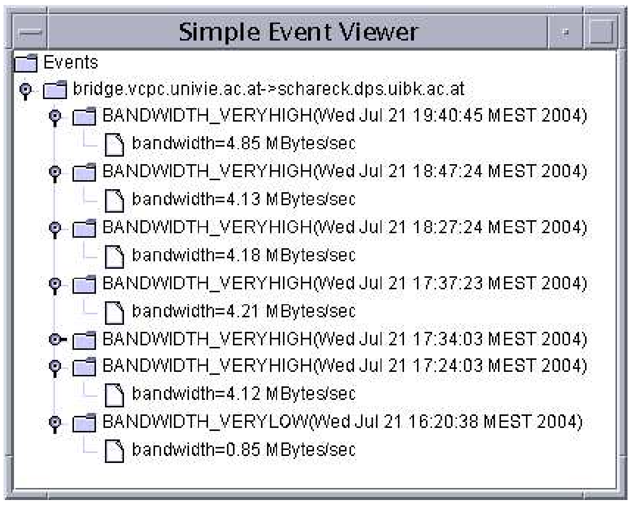
SCALEA–G: Application Profile Data Viewer

File Setting

```
Online Application Data
  Experiment:
    bridge.vcpc.univie.ac.at
      Process 0
        Thread 0
          Region 1:MainProgram[CR_P:0:0]
            Region 2:SubMyMain[CR_S:0:0]
              Region 4:MAIN_LOOP[CR_A:0:0]
                Region 6:SR_E_FIELD[CR_A:0:0]
                Region 8:SEND_RECV_BFIELD[C
                Region 9:IONIZATION[CR_A:0:0]
                Region 5:CAL_ION_YIELD[CR_A:
    gescher.vcpc.univie.ac.at
```

| Metric Name | Metric Value |
|---|---|
| ncalls | 458 |
| nsubs | 0 |
| wtime | 7,916,362 |
| utime | 1,973,693 |
| stime | 5,938,089 |
| majflt | 15 |
| minflt | 0 |
| nswap | 0 |

**Fig. 7.30.** Analysis of profiling data provided by application sensors.



Simple Event Viewer

```
Events
  bridge.vcpc.univie.ac.at->schareck.dps.uibk.ac.at
    BANDWIDTH_VERYHIGH(Wed Jul 21 19:40:45 MEST 2004)
      bandwidth=4.85 MBytes/sec
    BANDWIDTH_VERYHIGH(Wed Jul 21 18:47:24 MEST 2004)
      bandwidth=4.13 MBytes/sec
    BANDWIDTH_VERYHIGH(Wed Jul 21 18:27:24 MEST 2004)
      bandwidth=4.18 MBytes/sec
    BANDWIDTH_VERYHIGH(Wed Jul 21 17:37:23 MEST 2004)
      bandwidth=4.21 MBytes/sec
    BANDWIDTH_VERYHIGH(Wed Jul 21 17:34:03 MEST 2004)
    BANDWIDTH_VERYHIGH(Wed Jul 21 17:24:03 MEST 2004)
      bandwidth=4.12 MBytes/sec
    BANDWIDTH_VERYLOW(Wed Jul 21 16:20:38 MEST 2004)
      bandwidth=0.85 MBytes/sec
```

**Fig. 7.31.** Events generated from a rule-based sensor monitoring network bandwidth.

sensors), and of forecasting CPU usage and TCP bandwidth (forecasted data provided by demand-driven sensors). For example, data about CPU usage (waiting time, idle time, system time, user time) is measured per CPU. CPU monitoring data is periodically collected and the change of CPU usage can be observed on the fly through data subscription (see window *CPU Usage*). The machine `gescher` almost is idle whereas `bridge` is not fully utilized as only one of its four CPUs has high user time. Demand-driven sensors are used to obtain forecasted CPU usage and TCP bandwidth from NWS.

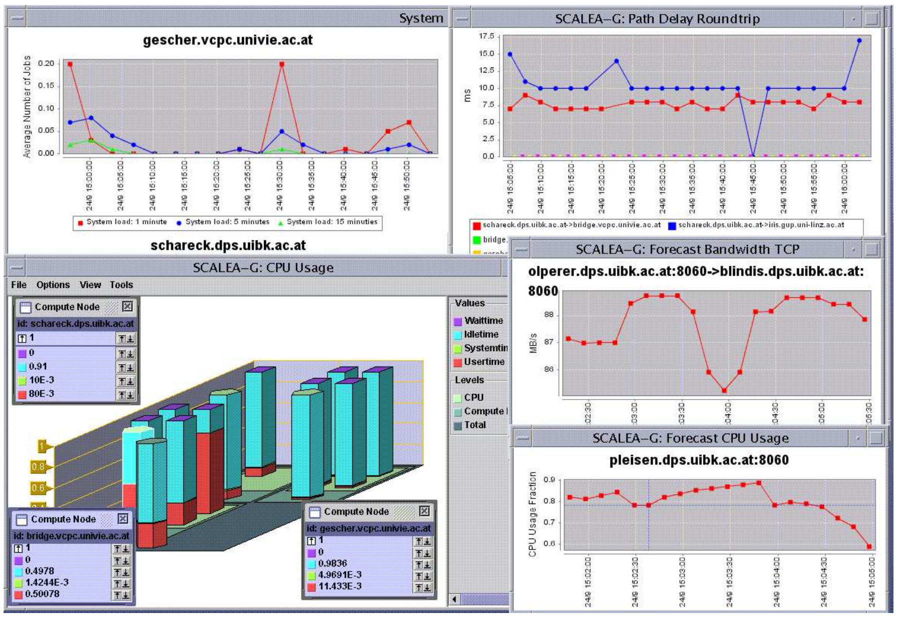

System
gescher.vcpc.univie.ac.at
Average Number of Jobs
System load: 1 minute
System load: 5 minutes
System load: 15 minutes
schareck.dps.uibk.ac.at
SCALEA-G: Path Delay Roundtrip
ms
schareck.dps.uibk.ac.at->bridge.vcpc.univie.ac.at
schareck.dps.uibk.ac.at->iris.gup.uni-linz.ac.at
SCALEA-G: CPU Usage
File Options View Tools
Compute Node
id: schareck.dps.uibk.ac.at
id: bridge.vcpc.univie.ac.at
id: gescher.vcpc.univie.ac.at
Values
Waittime
Idletime
Systemtime
Usertime
Levels
CPU
Compute
Total
SCALEA-G: Forecast Bandwidth TCP
olperer.dps.uibk.ac.at:8060->blindis.dps.uibk.ac.at:8060
MB/s
SCALEA-G: Forecast CPU Usage
pleisen.dps.uibk.ac.at:8060
CPU Usage Fraction

**Fig. 7.32.** Snapshots of online monitoring system load, CPU usage and networks.

## 7.4.6 Workflow Performance Monitoring and Analysis

In this section, we present experiments of different workflows of the Montage application in the Austrian Grid [28].

Montage [176] is a software for generating astronomical image mosaics with background modeling and rectification capabilities. Based on the Montage tutorial, we develop a set of WFs, each generates a mosaic from 10 images without applying any background matching. Figure 7.33 presents experimental workflows of the Montage application. In Figure 7.33(a), the activity `tRawImage` and `tUncorrectedMosaic` are used to transfer raw images from user site to computing site and resulting mosaics from computing site to user site, respectively. `mProject` is used to reproject input images to a common spatial scale. `mAdd` is used to coadd the reprojected images. `mImgtbl` is used to build image table which is accessed by `mProject` and `mAdd`.

In workflows executed on multiple resources, we have several subgraphs $tRawImage \rightarrow mImgtbl1 \rightarrow mProject1 \rightarrow tProjectedImage$, each subgraph is executed on a resource. The `tProjectedImage` activity is used to transfer projected images produced by `mProject` to the site on which `mAdd` is executed. When executed on $n$ resources, the subgraph $mImgtbl2 \rightarrow mAdd \rightarrow tUncorrectedMosaic$ is allocated on one of that $n$ resources. When executed on Grid resources using the same NFS (Network File System), the task `mProject` can work on fork-join fashion.

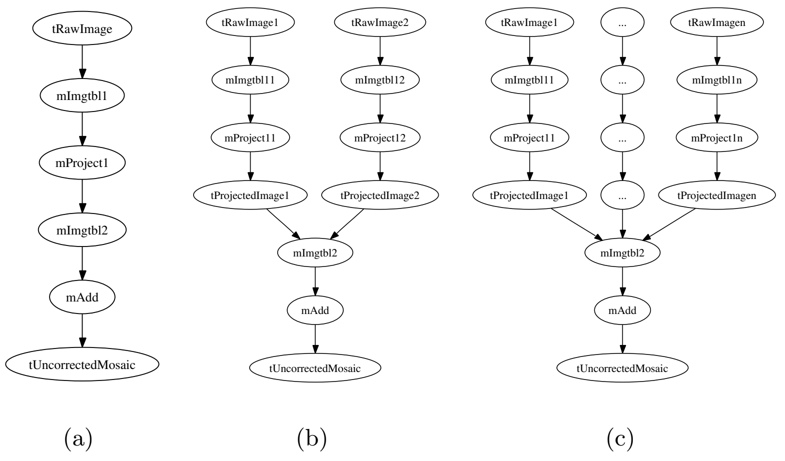


**Fig. 7.33.** Experimental workflows of the Montage application: (a) workflow executed on single resource, (b) workflow executed on two resources, and (c) workflow executed on $n$ resources.

We conduct experiments on sites named LINZ (Linz University), UIBK (University of Innsbruck), AURORA6 (University of Vienna) and VCPC (University of Vienna) of the Austrian Grid. The user resides in VCPC and the workflow invocation and control service (WIC) submits invoked applications of workflow activities to VCPC, LINZ, UIBK, AURORA6.

### 7.4.6.1 Monitoring Execution Status of Activities

Before a WF is submitted to WIC, the performance monitoring and analysis service subscribes notifications of workflow executions to the SCALEA-G middleware. When the WF is executed, events containing execution status (e.g. submitted, active, etc.) of activities are reported back to the monitoring and analysis service. Figure 7.34 shows the *Execution Status* display which monitors the execution status of activities. The left window shows one of Montage workflows. The right window displays execution status of activities of that workflow. We can also examine execution time of states during the runtime. For example, Figure 7.35 presents the execution time of states of the experiment presented in Figure 7.34.

### 7.4.6.2 Dynamic Instrumentation Example

When an activity is executed, its status is shown in the Execution Status diagram. The user then can start to instrument activity instances. Figure 7.36

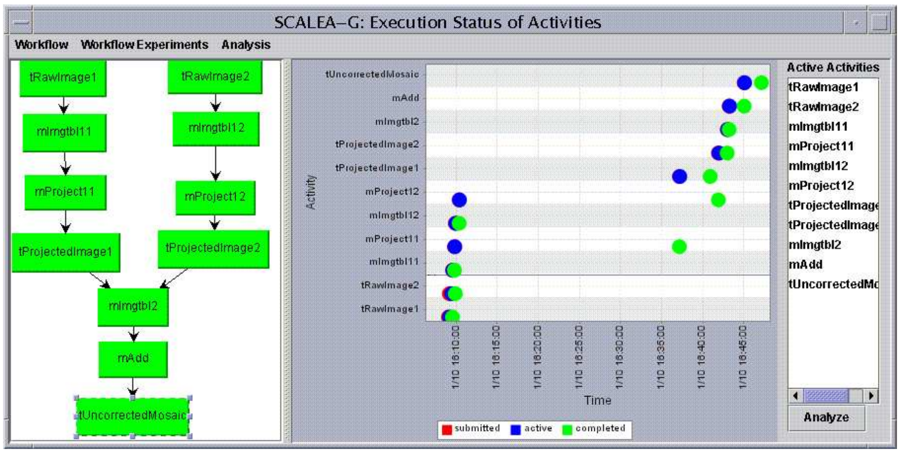


**Fig. 7.34.** Monitoring execution status of a Montage workflow executed on 2 resources.

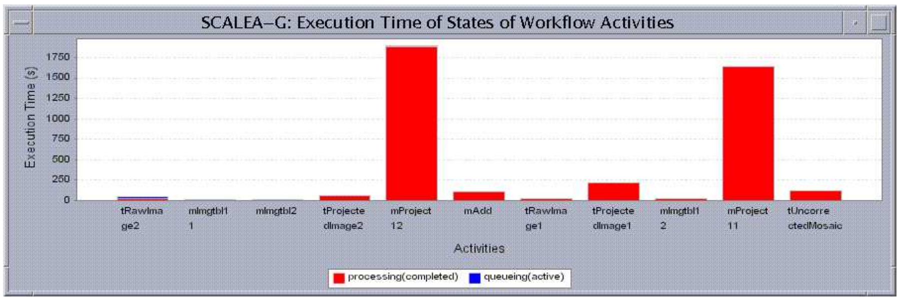
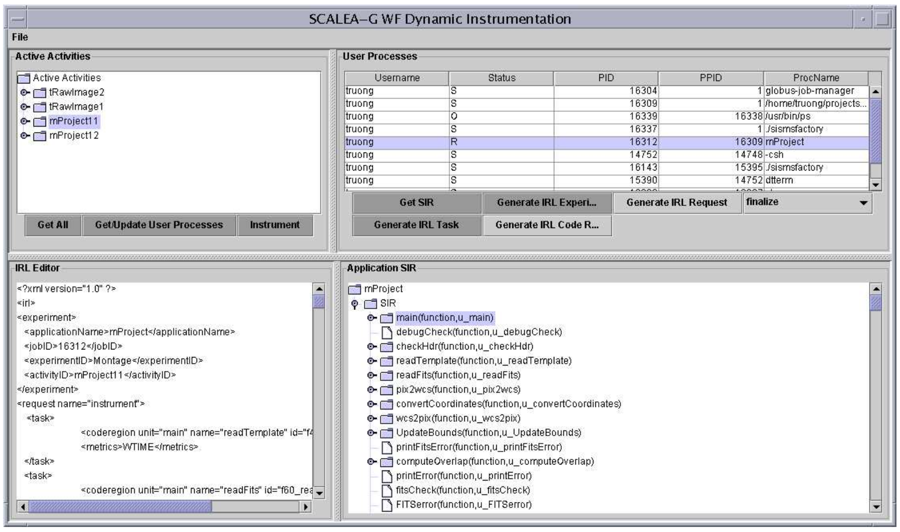


**Fig. 7.35.** Execution time of states of Montage workflow executed on 2 resources.

depicts the GUI used to control the dynamic instrumentation of activity instances. On the top-left window, the user can choose an activity. For each computational node on which the selected activity instance executed, active processes can be examined by invoking *GetUserProcesses* operation, as shown in the top-right window of Figure 7.36. For a given process of the invoked application of an activity instance, the detailed SIR can be obtained by clicking *GetSIR* button, e.g. SIR of invoked application of activity `mProject1` is visualized in the bottom-right window in Figure 7.36. In the bottom-left window is an IRL request used to instrument selected code regions in the `main` unit with a metric `wtime` (wall-clock time).

### 7.4.6.3 Performance Analysis

When an invoked application of an activity instance is instrumented, the measurement data collected is analyzed by performance analysis component. The



Active Activities
- tRawImage2
- tRawImage1
- mProject11
- mProject12

| Username | Status | PID | PPID | ProcName |
|---|---|---|---|---|
| truong | S | 16304 | 1 | globus-job-manager |
| truong | S | 16309 | 1 | /home/truong/projects... |
| truong | O | 16339 | 16338 | /usr/bin/ps |
| truong | S | 16337 | 1 | ./sismsfactory |
| truong | R | 16312 | 16309 | mProject |
| truong | S | 14752 | 14748 | -csh |
| truong | S | 16143 | 15395 | ./sismsfactory |
| truong | S | 15390 | 14752 | dtterm |

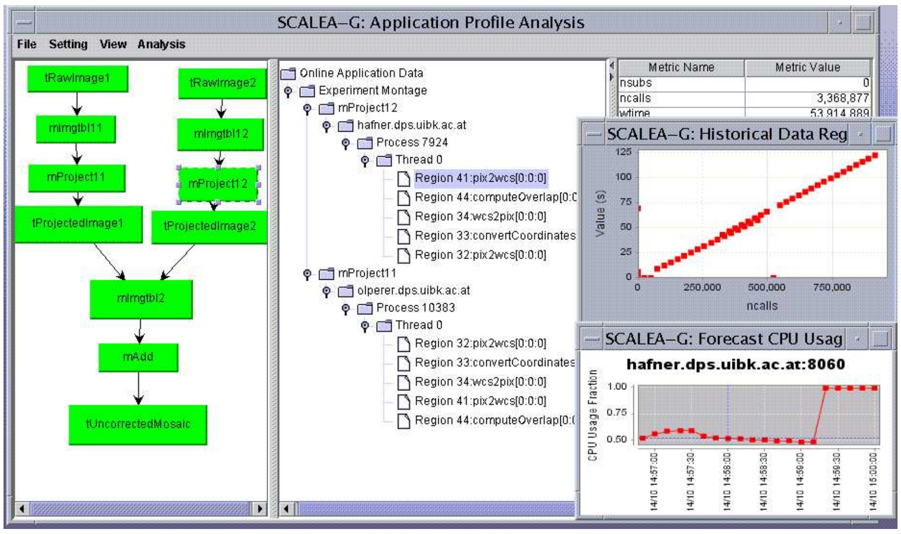
```
<?xml version="1.0" ?>
<irl>
<experiment>
  <applicationName>mProject</applicationName>
  <jobID>16312</jobID>
  <experimentID>Montage</experimentID>
  <activityID>mProject11</activityID>
</experiment>
<request name="instrument">
   <task>
                <coderegion unit="main" name="readTemplate" id="f4
                <metrics>WTIME</metrics>
  </task>
  <task>
                <coderegion unit="main" name="readFits" id="f60_rea
```

```
mProject
  SIR
    main(function,u_main)
    debugCheck(function,u_debugCheck)
    checkHdr(function,u_checkHdr)
    readTemplate(function,u_readTemplate)
    readFits(function,u_readFits)
    pix2wcs(function,u_pix2wcs)
    convertCoordinates(function,u_convertCoordinates)
    wcs2pix(function,u_wcs2pix)
    UpdateBounds(function,u_UpdateBounds)
    printFitsError(function,u_printFitsError)
    computeOverlap(function,u_computeOverlap)
    printError(function,u_printError)
    fitsCheck(function,u_fitsCheck)
    FITSerror(function,u_FITSerror)
```

**Fig. 7.36.** GUI used to control the instrumentation of activity instances of a workflow.

performance analysis component retrieves profiling data through data subscription or query. Figure 7.37 presents the performance analysis GUI when analyzing a Montage workflow executed on two resources in UIBK. The left-pane shows the DAG of the WF. The middle-pane shows the dynamic code region call graph (DRG) of invoked applications of activities. We can examine the profiling data of instrumented code region on the fly. The user can examine the whole DRG of the application, or DRG of an activity instance (by choosing the activity in the DAG). By clicking on a code region, detailed performance metrics will be displayed in the right-pane. Depending on the invoked application, source code information may be available, thus code regions can be associated with their sources. We can examine historical profiling data of a code region, for example window *Historical Data* shows the execution time of code region `computeOverlap` executed on `hafner.dps.uibk.ac.at`. The user can also monitor resources on which activities are executed. For example, the window *Forecast CPU Usage* shows the forecasted CPU usage of `hafner.dps.uibk.ac.at`.

Figure 7.38(a) presents the response time and synchronization delay analysis for activity *mImgtbl2* when the Montage workflow, presented in Figure 7.33(c), is executed on 5 machines, 3 of AURORA6 and 2 of LINZ. The synchronization delays from *tProjectedImage3, 4, 5* to *tImgtbl2* are very high. This causes by the high load imbalance between *mProject* instances, as shown in Figure 7.38(b). The load imbalance is not due to the inequality of work distribution between `mProject` activities, but due to the differences in processing capability of resources in the Grid. The two machines in LINZ can



| Metric Name | Metric Value |
|---|---|
| nsubs | 0 |
| ncalls | 3,368,877 |
| wtime | 53,914,889 |

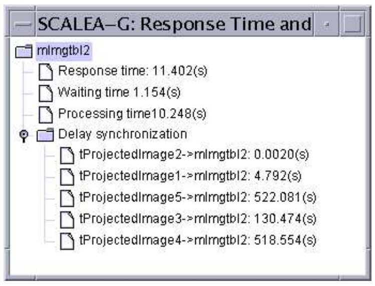
**Fig. 7.37.** Performance analysis of workflow activities.

process significantly faster than the rest machines in AURORA6. This detection indicates the workflow composition system and scheduling system do not take into account the processing capability of resources when constructing activities and distributing them on Grids.

Over the course of workflow development, subgraph named `mProjectedImage` which includes $tRawImage \rightarrow mImgtbl1 \rightarrow mProject1$ in single resource version is replaced by subgraphs of $tRawImage \rightarrow mImgtbl1 \rightarrow mProject1 \rightarrow tProjectedImage$ in a multi-resource version. These subgraphs basically provide projected images to the `mAdd` activity, therefore, we consider they are equivalent in terms of QoS (to the user point of view); they are replaced refinement graphs. We collect and store performance of these subgraphs in different experiments. Figure 7.39 shows the speedup factor for the subgraph `mProjectedImage` of Montage workflows executed on several experiments. The execution of `mProjectedImage` of the workflow executed on single resource in LINZ is faster than that of its refinement graph executed on two resources (in AURORA6, or UIBK). However, the execution of `mProjectedImage` of workflow executed on 5 resources, 3 of AURORA6 and 2 of LINZ, is just very slightly faster than that executed on 5 resources of AURORA6. The reason is that the slower activities executed on AURORA6 resources have a significant impact on the overall execution of the whole `mProjectedImage`, as presented on Figure 7.38(b).

### 7.4.7 Searching Example with PERFONTO

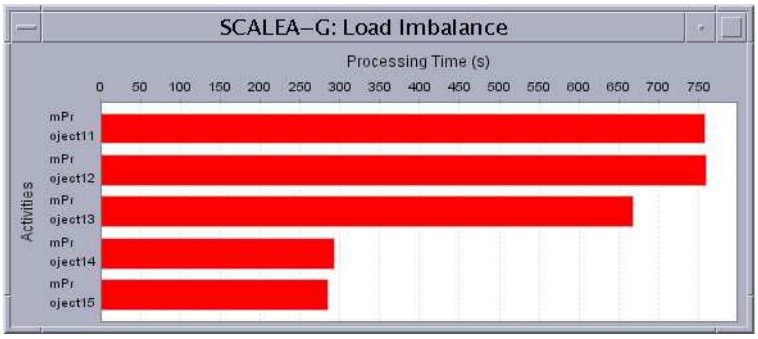
In this section, we present an example of using our current prototype to search ontology-based performance data. We used SCALEA to conduct the instru-



SCALEA–G: Response Time and

mImgtbl2
- Response time: 11.402(s)
- Waiting time 1.154(s)
- Processing time10.248(s)
- Delay synchronization
  - tProjectedImage2->mImgtbl2: 0.0020(s)
  - tProjectedImage1->mImgtbl2: 4.792(s)
  - tProjectedImage5->mImgtbl2: 522.081(s)
  - tProjectedImage3->mImgtbl2: 130.474(s)
  - tProjectedImage4->mImgtbl2: 518.554(s)

(a)



(b)

**Fig. 7.38.** Analysis of Montage executed on 5 machines: (a) response time and synchronization delay of `mImgtbl`, and (b) load imbalance of `mProject`.

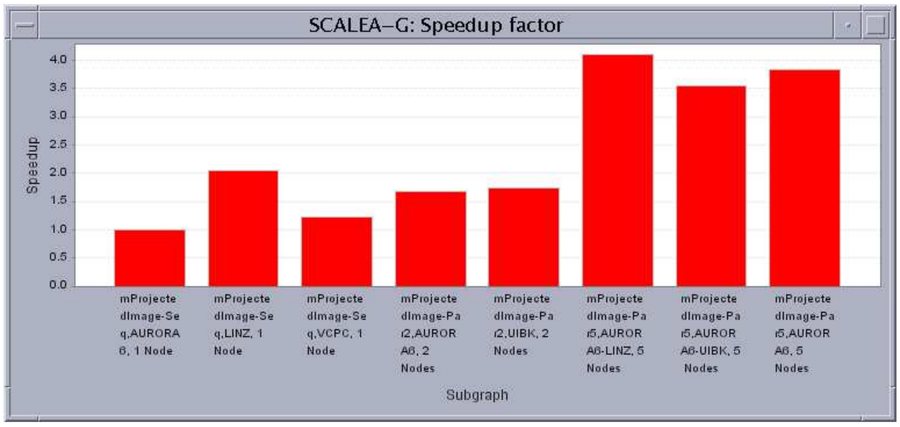
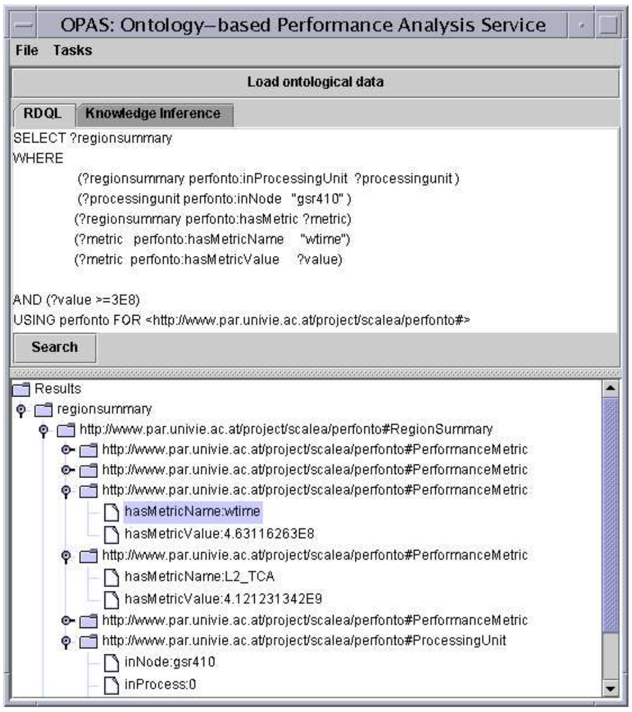


**Fig. 7.39.** Speedup factor for subgraph `ProjectedImage` of Montage workflows.

mentation and measurement for the 3D Particle-In-Cell application (3DPIC) [102] in a cluster of SMPs and stored performance data into the repository service.



```
SELECT ?regionsummary
WHERE
          (?regionsummary perfonto:inProcessingUnit ?processingunit)
          (?processingunit perfonto:inNode  "gsr410")
         (?regionsummary perfonto:hasMetric ?metric)
         (?metric  perfonto:hasMetricName   "wtime")
         (?metric  perfonto:hasMetricValue   ?value)

AND (?value >=3E8)
USING perfonto FOR <http://www.par.univie.ac.at/project/scalea/perfonto#>
```

```
Results
  regionsummary
    http://www.par.univie.ac.at/project/scalea/perfonto#RegionSummary
      http://www.par.univie.ac.at/project/scalea/perfonto#PerformanceMetric
      http://www.par.univie.ac.at/project/scalea/perfonto#PerformanceMetric
      http://www.par.univie.ac.at/project/scalea/perfonto#PerformanceMetric
        hasMetricName:wtime
        hasMetricValue:4.63116263E8
      http://www.par.univie.ac.at/project/scalea/perfonto#PerformanceMetric
        hasMetricName:L2_TCA
        hasMetricValue:4.121231342E9
      http://www.par.univie.ac.at/project/scalea/perfonto#PerformanceMetric
      http://www.par.univie.ac.at/project/scalea/perfonto#ProcessingUnit
        inNode:gsr410
        inProcess:0
```

**Fig. 7.40.** Example of search on ontological data

We use OPAS GUI to connect the repository service to retrieve ontology models and instance data. Figure 7.40 presents a user interface for conducting performance search of ontological data. In the top window the user can specify searching queries whereas the result will be shown in the bottom window. For example, we conducted a search with the query presented in Section 6.5.4.2. In the bottom window, under the sub tree of variable *regionsummary*, a list of region summaries met the condition will be shown. The user can examine performance metrics in detail. Also other information such as source code and machine can be visualized as needed.

## 7.5 Summary

We have presented a variety of experiments of monitoring, instrumentation, and performance analysis of real-world applications for cluster and Grid systems. These experiments presents the novelty and usefulness of our methods and techniques for assisting the user to monitor, instrument, measure and analyze the performance of cluster and Grid applications developed in various programming models. Moreover, we have presented other novel features of SCALEA and SCALEA-G to support the monitoring, instrumentation, and performance analysis such as self-management monitoring middleware, rule-based monitoring, dynamic instrumentation for Grid applications, and ontology-based performance search.

As SCALEA/SCALEA-G provide instrumentation, measurement and performance analysis facilities for other tools in ASKALON toolset, a large set of experiments that partially illustrates the novelty and usefulness of SCALEA and SCALEA-G can be found in [200, 217, 87, 85, 84, 119]. Although these experiments demonstrate functionalities of corresponding tools, e.g., parameter study, bottleneck search, performance prediction, these experiments partially reflect the usefulness of SCALEA/SCALEA-G because the corresponding tools base on SCALEA/SCALEA-G for performance instrumentation, measurement and analysis of applications.

# 8 Conclusion and Future Work

## 8.1 Conclusion

The rapid evolution of hardware and underlying system models (e.g. from cluster SMP to the Grid) and programming paradigms (e.g. from cluster-based OpenMP/MPI applications to Grid workflow-based applications) for high performance computing results in many challenges to be solved by the performance analysis community. On the one hand, performance tools have to be adjusted, redesigned, and developed for new systems and programming paradigms. On the other hand, new monitoring, instrumentation and analysis techniques and methods need to be developed in order to reveal and analyze new performance metrics, properties and models of the new programming paradigms and systems. The application appears to be more complex; also the performance tuning and testing are more intensive, large-scale and long-running. As a result, performance tools have to collect, store and analyze a large amount of performance data for the performance analysis and tuning. New techniques and methods for storing and querying and analyzing such large amount of data are required.

To address the above-mentioned challenges, one can focus on an isolated issue, e.g. instrumentation or monitoring or analysis. However, performance monitoring, instrumentation and analysis are related issues which must be tackled together because they are interrelated. This dissertation addresses some of the above-mentioned challenges for cluster and Grid systems. The research approach we follow is that we consider performance monitoring, instrumentation, and analysis as a monolithic system and we always tackle them in an integrated manner. Another aspect is that we consider storing and sharing performance data and providing a well-defined interface for other tools and services to utilize the performance tool and data to be key issues for performance analysis tools. These issues are not on the focus of most existing performance analysis tools.

In the first part of this dissertation, we have presented novel techniques for execution-driven performance analysis of cluster applications. We have described a new classification of performance overheads for shared and distributed memory parallel programs. The overhead classification provides a detailed analysis of the sources of performance overheads for parallel programs. A highly flexible instrumentation system can be used to instrument arbitrary code regions and enables the programmer to request for a large variety of performance metrics ranging from timing information and hardware (HW) parameters to performance overheads. The overhead classification is used as an input for the instrumentation in detecting and selecting code regions which are likely the causes of performance problems. While in most existing tools the task of instrumentation is solely targeted to the end-user, we also provide a high-level instrumentation interface based on an abstract syntax tree and an unparser provided by an external Fortran90 compiler. The instrumentation interface can be used by other higher-level tools for instrumenting parallel and distributed Fortran programs, e.g. by a bottleneck search tool. A novel representation for code regions named dynamic code region call graph (DRG) has been introduced. The DRG reflects the dynamic relationship between code regions and its subregions and enables a detailed overhead analysis for every code region. The DRG is not restricted to function calls but covers every code regions ranging from single statements to entire program units.

We designed and implemented an experiment data repository that holds all relevant information about experiments including application source information and performance experiments. The data repository is capable of exporting experiment data in XML format and simplifies interactions among performance analysis tools and high-level tools by providing a well-defined data schema and interface for accessing information stored in the repository. By utilizing the performance data repository we can go beyond the analysis of one experiment at a time by supporting performance analysis of multiple experiments.

The overhead analysis, customizable instrumentation, experiment data repository, and multi-experiment analysis, have been implemented in SCALEA framework. SCALEA is among the first performance analysis tools that combines source code and HW profiling in a single system, supporting multiple parallel program models including OpenMP, MPI, HPF and hybrid paradigms (e.g., OpenMP/MPI), and providing multi-experiment analysis.

We have presented a novel approach to performance analysis, the soft performance analysis, which is based on soft computing model. Proposed soft performance analysis techniques exploit and combine fuzzy logic and similarity theory, and machine learning algorithms to provide soft, scalable and intelligent techniques for analyzing and comparing the performance of large and complex parallel and distributed applications. We contribute flexible and convenient methods to deal with uncertainty in performance analysis (e.g., fuzzy-based bottleneck search), means for conducting performance analysis in the form closer to human notions (e.g., fuzzy-based search query). A prototype of soft performance analysis techniques has been developed and integrated into the SCALEA framework.

In the second part of this dissertation, we have presented a unified system for performance monitoring and analysis for the Grid. We presented a novel sensor-based middleware for performance monitoring and data integration in Grids; the middleware is capable of self-management. Our framework has many novelties. Firstly, the framework unifies different types of sensors including system and application sensors for monitoring and performance analysis in a single system, and integrates various types of data from many sources. Secondly, the utilization of both Grid service-based operation and TCP-based data stream communication allows us to take advantages of interoperability and manageability and to balance tradeoffs between the interoperability and the performance. Our middleware stores collected data on distributed sites and provides the same mechanism for accessing that distributed data, thus increasing not only the scalability but also the fault-tolerance. By incorporating P2P technologies, our middleware is capable of supporting data discovery and data relay for service community. As a result, it helps to cope with limitations due to firewall, and increases availability and fault-tolerance.

We have introduced a novel Grid service of dynamic instrumentation for Grid applications. We present a new Grid performance analysis service that can be used to monitor and analyze the performance of scientific workflows in the Grid on the fly. The Grid performance analysis service, which combines dynamic instrumentation, activity execution monitoring, and performance analysis of workflows in a single system, has presented a dynamic and flexible way that supports the user to monitor and analyze their applications. Moreover, we store workflows and their relevant performance metrics and develop techniques for comparing the performance of subgraphs of workflows, and for supporting multi-workflow analysis.

The self-managing sensor-based middleware has been implemented in SCALEA-G. Grid dynamic instrumentation and performance analysis service have been implemented and integrated into SCALEA-G. SCALEA-G is one of the first OGSA-based Grid monitoring and analysis systems for the Grid. SCALEA-G is unique by unifying both infrastructure and application monitoring in a single system. To our best knowledge, there is no other Grid dynamic instrumentation service that offers a widely accessible interface through Web/Grid service.

We have described how ontology can help to overcome the lack of semantics possessed by current techniques used in existing performance monitoring and measurement tools to describe performance data in Grid. We have developed an ontology for representing performance data along with a Grid service for archiving and providing ontology models and performance data. Initial results show that ontology is a promising solution in the domain of performance analysis because it not only provides a common understanding about performance data for tools and services but also increases the automation of performance analysis.

It is worth mentioning that our techniques and methods presented in this dissertation are the results of a research curve in an incremental manner. We devise a technique to address an existing problem, and when working with this technique, the solution of the technique raises new problems. Subsequently, it comes up with a new technique that can address the new problems. For example, we first developed crisp search of performance data on the relational database, and then figured out that an approximate search is needed. That motivated the proposal and development of fuzzy-based search. Or we first stored performance data of parallel programs into relational databases, but as the Grid requires widely accessible means to access performance data, it turned out that performance data should be described in XML. However, we then realized that XML may not be suitable for Grid performance data. That motivated the proposed ontological performance data. The new techniques and methods do not eliminate the old ones, rather they complement each other. The important rule is that we should use the appropriate technique for appropriate applications and systems.

## 8.2 Future Work

### 8.2.1 Soft Performance Analysis

Although we have extensively studied and applied various membership and distance functions for performance score and similarity, these functions are very much dependent on specific metrics on specific systems for specific applications. Thus, we need to conduct a comprehensive evaluation of membership functions which are used to measure performance score and of distance functions which are used to measure performance similarity. In our experiments we employ the most common membership functions, but by utilizing analytical and simulation techniques, we may obtain more suitable membership functions.

Still the soft performance analysis approach is just at an early stage, we believe that it is a promising solution to support the specification and the control of approximate and inexact parameters in performance analysis, and to provide soft, scalable and intelligent methods for handling current large amounts of performance data which may contain sources of uncertainty. We have just applied our proposed techniques to analyze performance data of parallel programs in cluster environments, however, these techniques could be applied to performance analysis of Grid environments. On the Grid, resources and their usage are unpredictable, and performance data collected tends to be more imprecision and uncertainty than those collected on cluster environments. Moreover, performance similarity can be used to analyze and compare diverse Grid resources. The resultant performance similarities and differences can be used as inputs to the resource matching and selection in the Grid.

Currently, we compute and compare similarity measures of experiment factors and the performance of code regions. However, we do not establish any model to illustrate the interaction between these measures. The future work is to study the fuzzy rules among experiment factors and the performance of code regions.

### 8.2.2 Middleware for Grid Monitoring and Data Integration

Although P2P and autonomic features give many promising solutions to solve challenges in Grid monitoring, it is not a simple task to incorporate these features into a middleware. We continue to develop and exploit these features as part of a middleware for Grid monitoring and performance data integration. To continue our effort on utilizing sensor networks, P2P and autonomic computing features, the set of sensors will be extended, together with effectors, to support self-healing. We plan to provide adaptive capabilities for services of our middleware that they can self-adjust their functions under the computing capabilities of the hosting environment. Besides, we also investigate the support of monitoring based on ontological performance data.

### 8.2.3 Grid Workflow Instrumentation and Analysis

In the current implementation, there is a lack of connections between the workflow management system and our monitoring and performance analysis middleware. Therefore, we have to manually instrument the concrete workflow code in order to get information about the execution of activities and their relationship. To avoid that, we can extend workflow specification language with directives used to obtain these information. These directives will be translated into code used to publish events into the monitoring middleware.

Currently our supported workflows limit to DAGs. However, there is a growing trend of developing Grid scientific workflows which contain structured loops (e.g. do while). The performance analysis for workflows will be extended to cover the workflows having loops. Moreover, performance analysis for Grid workflows has to be extended to monitor and analyze other metrics, e.g. coordination constraints, rather than only timing and hardware metrics at meanwhile.

Another aspect is that while we focus on invoked applications as executable programs (each activity instance invokes an executable program), there exist workflows whose an activity instance invokes a Web Service operation, which may wrap executable programs. This type of workflows will require different instrumentation mechanisms, e.g. dynamic instrumentation of Java services. Meanwhile, the process of analysis, monitoring and instrumentation is controlled by the end-user, but this process will be automated.

IRL will be extended to support the specification of more complex instrumentation requests such as events, the activation and deactivation of the instrumentation.

### 8.2.4 Semantic Performance Data and Ontology-based Performance Analysis

We will continue our effort on investigating and applying ontology for performance analysis. In the field of developing semantic representation for performance data, we are currently enhancing and reevaluating our proposed ontology in many directions. One of the directions is that we consider to separate PERFONTO into two separate ontology models for class descriptions: *experiment-related class* (related to application and its execution behavior, e.g., performance metrics, events) and *system-related class* (related to computing resources allocated for executing experiments, e.g., clusters, networks, computational nodes). By doing so, we can easily extend ontology models. Another direction is to extend the ontology to describe performance properties and performance data of workflow applications. We are currently studying performance metrics and developing ontology for describing performance data of workflows [256]. In addition, we plan to develop a task-based ontology that describes conceptualizations of tasks and tools along with their interrelationships and properties for an automatic performance analysis system [105].

# A

# Notations

| Symbol | Description |
|---|---|
| $\in$ | set membership |
| $\notin$ | not set membership |
| $\{x_1, x_2, \cdots, x_n\}$ | a set includes $x_1, x_2, \cdots, x_n$ |
| $\wedge$ | logical conjunction |
| $\vee$ | logical disjunction |
| $\exists$ | exists |
| $\cap$ | intersection |
| $\cup$ | union |
| $\equiv$ | equivalence |
| $\sum$ | sum |
| $\forall$ | for all |
| $A^T$ | the transpose of matrix/vector A |
| $m$ | metric |
| $T_o$ | temporal overhead |
| $T_o^i$ | temporal overhead of thread $i$ |
| $T_o(r)$ | temporal overhead of code region $r$ |
| $T_o^i(r)$ | temporal overhead of code region $r$ executed on processor $i$ |
| $T_i$ | identified overhead |
| $T_u$ | unidentified overhead |
| $T_p$ | execution time of parallel version with $p$ processors |
| $T_s$ | execution time of sequential version |
| $T_p(r)$ | execution time of code region $r$ with $p$ processors |
| $T^i(r)$ | execution time of code region $r$ on processor numbered $i$ |
| $T_{name}(r)$ | performance metric *name* (in time) of code region $r$ |
| $T^i_{name}(r)$ | performancemetric *name* of code region $r$ in processor $i$ |
| $c_{r_{m_i}}$ | activation of code region $r_{m_i}$ |
| $c^l_{r_i} \rightarrow c^k_{r_j}$ | activation numbered $k$ of code region $r_j$ is called directly by activation numbered $l$ of code region $r_i$ |
| $pu$ | processing unit |

| **Symbol** | **Description** |
|---|---|
| $\mu(x)$ | membership function that maps $x$ into range $[0,1]$ |
| $s_i$ | performance score |
| $OWA(\vec{s})$ | overall weighted average score of a score vector |
| $\vec{s}$ | a vector of performance scores |
| $rs$ | region summary |
| $sim(rs_i, rs_j)$ | similarity measure between region summary $rs_i$ and $rs_j$ |
| $sim_f(e_i, e_j)$ | similarity measure between factor $f$ in experiment $e_i$ and $e_j$ |
| $d_{ij}$ | distance function between two scores |
| $d_{fcm}$ | distance function used in fuzzy C-means |
| $(a_i, a_j)$ | the dependency between activity $a_i$ and $a_j$ |
| *pred($a_i$)* | set of the immediate predecessors of $a_i$ |
| *succ($a_i$)* | set of the immediate successors of $a_i$ |
| $P(a)$ | activity execution status graph of activity $a$ |
| $(e, s)$ | the leading event $e$ of state $s$ |
| $(s, e)$ | the ending event $e$ of state $s$ |
| $t_{now}$ | timestamp of the current time |
| $e_i \rightarrow e_j$ | event $e_i$ happens before event $e_j$ |
| $e_{name}(a)$ | an event *name* of $P(a)$ |
| $t(e)$ | the timestamp of an event $e$ |
| $T_{sd}(a_i, a_j)$ | the synchronization delay from $a_i$ to $a_j$ |
| `1Nx4P` | 1 SMP node, 4 processors per node |
| `P4` | MPICH P4 |
| `GM` | MPICH Myrinet |

# B

# Abbreviations and Accronyms

| **Abbreviation** | **Description** |
|---|---|
| ACL | Access Control List |
| ANOVA | Analysis of Variance |
| CPU | Central Processing Unit |
| CR | Code Region |
| CS | Client Service |
| DPG | Dynamic Processing Unit Graph |
| DQS | Data Query and Subscription |
| DRG | Dynamic Code Region Call Graph |
| DS | Directory Service |
| FL | Fuzzy Logic |
| FS | Fuzzy Set |
| GSI | Grid Security Infrastructure |
| GUI | Graphical User Interface |
| HPC | High Performance Computing |
| HPF | High Performance Fortran |
| HW | Hardware |
| IL | Intermediate Language |
| IRL | Instrumentation Request Language |
| LAN | Local Area Network |
| MF | Mutator Factory |
| MFLOPS | Millions Floating-Point Operations Per Second |
| MI | Mutator Instance |
| ML | Machine Learning |
| MPI | Message Passing Interface |
| MPP | Massive Parallel Processing |
| MS | Mutator Service |
| OS | Operating System |
| OpenMP | Open Multi-Processing |
| OWA | Overall Weighted Average |

| Abbreviation | Description |
|---|---|
| OWL | Web Ontology Language |
| PC | Personal Computer |
| PERFONTO | Ontology for Performance Analysis |
| PVM | Parallel Virtual Machine |
| PVP | Parallel Vector Processing |
| PERFQL | Performance Query Language based on Fuzzy Logic |
| P2P | Peer-to-Peer |
| RMA | Remote Memory Access |
| SIS | SCALEA Instrumentation System |
| SIR | Standardized Intermediate Representation |
| SIRBC | Standardized Intermediate Representation for Binary Code |
| SM | Sensor Manager Service |
| SMP | Symmetric Multi-Processor |
| VFC | Vienna Fortran Compiler |
| WF | Workflow |
| WFA | Workflow-based Application |
| WfMS | Workflow Management System |

C

# Code Region and Performance Metric Mnemonics

## C.1 Code Region Mnemonics

Code region mnemonics are used to specify types of code regions. Code region mnemonics are listed in Table C.1.

| **Mnemonics** | **Descriptions** |
|---|---|
| CR_P | Main program |
| CR_A | Arbitrary code region |
| CR_L | All loops (general) |
| CR_U | Outermost loop |
| CR_B | Branch code region |
| CR_W | I/O Write operation |
| CR_R | I/O read operation |
| CR_O | I/O open operation |
| CR_C | I/O close operation |
| CR_Y | Procedure's internal (function or subroutine) |
| CR_S | Subroutine call |
| CR_F | Function calls |
| CR_COMALL | All common code regions |
| CR_I | INDEPENDENT loop (HPF) |
| CR_D | Work distribution (HPF) |
| CR_N | Inspector (HPF) |
| CR_X | Executor (HPF) |
| CR_G | Gather (HPF) |
| CR_T | Scatter (HPF) |
| CR_HPFALL | All HPF code regions |
| CR_OMPPA | OMP PARALLEL |
| CR_OMPPD | OMP PARALLEL DO |
| CR_OMPPS | OMP PARALLEL SECTIONS |
| CR_OMPPW | OMP PARALLEL WORKSHARE |
| CR_OMPDO | OMP DO |

| **Mnemonics** | **Descriptions** |
|---|---|
| CR_OMPSE | OMP SECTIONS |
| CR_OMPWO | OMP WORKSHARE |
| CR_OMPSI | OMP SINGLE |
| CR_OMPMA | OMP MASTER |
| CR_OMPBA | OMP BARRIER |
| CR_OMPCR | OMP CRITICAL |
| CR_OMPAT | OMP ATOMIC |
| CR_OMPOR | OMP ORDERED |
| CR_OMPFL | OMP FLUSH |
| CR_OMPSE | OMP SECTION |
| CR_OMPICR | Codes inside OMP CRITICAL |
| CR_OMPIOR | Codes inside OMP ORDERED |
| CR_OMPISE | Codes inside OMP SINGLE |
| CR_OMPBPA | OMP PARALLEL directive |
| CR_OMPEPA | OMP END PARALLEL directive |
| CR_OMPIDO | The body of do loop of OMP DO |
| CR_OMPLO | OpenMP locks |
| CR_OMPALL | All OpenMP code regions |
| CR_MPISTARTUP | MPI initialization and MPI finalization |
| CR_MPIP2P | MPI point-to-point communication |
| CR_MPISEND | MPI Send |
| CR_MPIRECV | MPI Receive |
| CR_MPICOL | MPI Collective communication |
| CR_MPITP | MPI data type conversions |
| CR_MPIBA | MPI barrier |
| CR_MPIALL | All MPI code regions |
| CR_OTHERREP | Replicated code region |
| CR_OTHERSEQ | Sequential code region |

Table C.1: Code region mnemonics

## C.2 Performance Metric Mnemonics

Mnemonics of performance metrics are classified into mnemonics for measured timing, measured counter (including hardware counters) and overhead metrics. Table C.2 presents mnemonics of measured timing and counter metrics. The mnemonics for hardware counter metrics are based on hard counter names supported by PAPI [49]. These names are alike to those in PAPI (in fact similar to those supported by current hardware vendors) because our hardware measurement is based on PAPI. However, as it is possible to interface our tool to other hardware measurement libraries, we opt to change slightly PAPI metric names. Mnemonics for overhead metrics are presented in Table C.3.

| **Mnemonics** | **Data Type** | **Unit** | **Descriptions** |
|---|---|---|---|
| wtime | double | microsecond | Wall-clock time |
| utime | double | microsecond | User CPU time |
| stime | double | microsecond | System CPU time |
| ctime | double | microsecond | CPU Time, including user CPU time and system CPU time |
| ncalls | int64 | counter | Number of executions |
| nsubs | int64 | counter | Number of executions of sub code regions |
| majflt | int64 | counter | Page faults requiring physical I/O |
| minflt | int64 | counter | Page faults not requiring physical I/O |
| nswap | int64 | counter | Number of swaps |
| L1_DCM | int64 | counter | Level 1 data cache misses |
| L1_ICM | int64 | counter | Level 1 instruction cache misses |
| L2_DCM | int64 | counter | Level 2 data cache misses |
| L2_ICM | int64 | counter | Level 2 instruction cache misses |
| L3_DCM | int64 | counter | Level 3 data cache misses |
| L3_ICM | int64 | counter | Level 3 instruction cache misses |
| L1_TCM | int64 | counter | Level 1 cache misses |
| L2_TCM | int64 | counter | Level 2 cache misses |
| L3_TCM | int64 | counter | Level 3 cache misses |
| CA_SNP | int64 | counter | Requests for a snoop |
| CA_SHR | int64 | counter | Requests for exclusive access to shared cache line |
| CA_CLN | int64 | counter | Requests for exclusive access to clean cache line |
| CA_INV | int64 | counter | Requests for cache line invalidation |
| CA_ITV | int64 | counter | Requests for cache line intervention |
| L3_LDM | int64 | counter | Level 3 load misses |
| L3_STM | int64 | counter | Level 3 store misses |
| BRU_IDL | int64 | counter | Cycles branch units are idle |
| FXU_IDL | int64 | counter | Cycles integer units are idle |
| FPU_IDL | int64 | counter | Cycles floating point units are idle |
| LSU_IDL | int64 | counter | Cycles load/store units are idle |
| TLB_DM | int64 | counter | Data translation lookaside buffer misses |
| TLB_IM | int64 | counter | Instruction translation lookaside buffer misses |
| TLB_TL | int64 | counter | Total translation lookaside buffer misses |
| L1_LDM | int64 | counter | Level 1 load misses |
| L1_STM | int64 | counter | Level 1 store misses |
| L2_LDM | int64 | counter | Level 2 load misses |

| Mnemonics | Data Type | Unit | Descriptions |
|---|---|---|---|
| L2_STM | int64 | counter | Level 2 store misses |
| BTAC_M | int64 | counter | Branch target address cache misses |
| PRF_DM | int64 | counter | Data prefetch cache misses |
| L3_DCH | int64 | counter | Level 3 data cache hits |
| TLB_SD | int64 | counter | Translation lookaside buffer shootdowns |
| CSR_FAL | int64 | counter | Failed store conditional instructions |
| CSR_SUC | int64 | counter | Successful store conditional instructions |
| CSR_TOT | int64 | counter | Total store conditional instructions |
| MEM_SCY | int64 | counter | Cycles Stalled Waiting for memory accesses |
| MEM_RCY | int64 | counter | Cycles Stalled Waiting for memory Reads |
| MEM_WCY | int64 | counter | Cycles Stalled Waiting for memory writes |
| STL_ICY | int64 | counter | Cycles with no instruction issue |
| FUL_ICY | int64 | counter | 26 Cycles with maximum instruction issue |
| STL_CCY | int64 | counter | Cycles with no instructions completed |
| FUL_CCY | int64 | counter | Cycles with maximum instructions completed |
| HW_INT | int64 | counter | Hardware interrupts |
| BR_UCN | int64 | counter | Unconditional branch instructions |
| BR_CN | int64 | counter | Conditional branch instructions |
| BR_TKN | int64 | counter | Conditional branch instructions taken |
| BR_NTK | int64 | counter | Conditional branch instructions not taken |
| BR_MSP | int64 | counter | Conditional branch instructions mispredicted |
| BR_PRC | int64 | counter | Conditional branch instructions correctly predicted |
| FMA_INS | int64 | counter | FMA instructions completed |
| TOT_IIS | int64 | counter | Instructions issued |
| TOT_INS | int64 | counter | Instructions completed |
| INT_INS | int64 | counter | Integer instructions |
| FP_INS | int64 | counter | Floating point instructions |
| LD_INS | int64 | counter | Load instructions |
| SR_INS | int64 | counter | Store instructions |
| BR_INS | int64 | counter | Branch instructions |
| VEC_INS | int64 | counter | Vector/SIMD instructions |

| **Mnemonics** | **Data Type** | **Unit** | **Descriptions** |
|---|---|---|---|
| FLOPS | int64 | counter | Floating point instructions per second |
| RES_STL | int64 | counter | Cycles stalled on any resource |
| FP_STAL | int64 | counter | Cycles the FP unit(s) are stalled |
| TOT_CYC | int64 | counter | Total cycles |
| IPS | int64 | counter | Instructions per second |
| LST_INS | int64 | counter | Load/store instructions completed |
| SYC_INS | int64 | counter | Synchronization instructions completed |
| L1_DCH | int64 | counter | Level 1 data cache hits |
| L2_DCH | int64 | counter | Level 2 data cache hits |
| L1_DCA | int64 | counter | Level 1 data cache accesses |
| L2_DCA | int64 | counter | Level 2 data cache accesses |
| L3_DCA | int64 | counter | Level 3 data cache accesses |
| L1_DCR | int64 | counter | Level 1 data cache reads |
| L2_DCR | int64 | counter | Level 2 data cache reads |
| L3_DCR | int64 | counter | Level 3 data cache reads |
| L1_DCW | int64 | counter | Level 1 data cache writes |
| L2_DCW | int64 | counter | Level 2 data cache writes |
| L3_DCW | int64 | counter | Level 3 data cache writes |
| L1_ICH | int64 | counter | Level 1 instruction cache hits |
| L2_ICH | int64 | counter | Level 2 instruction cache hits |
| L3_ICH | int64 | counter | Level 3 instruction cache hits |
| L1_ICA | int64 | counter | Level 1 instruction cache accesses |
| L2_ICA | int64 | counter | Level 2 instruction cache accesses |
| L3_ICA | int64 | counter | Level 3 instruction cache accesses |
| L1_ICR | int64 | counter | Level 1 instruction cache reads |
| L2_ICR | int64 | counter | Level 2 instruction cache reads |
| L3_ICR | int64 | counter | Level 3 instruction cache reads |
| L1_ICW | int64 | counter | Level 1 instruction cache writes |
| L2_ICW | int64 | counter | Level 2 instruction cache writes |
| L3_ICW | int64 | counter | Level 3 instruction cache writes |
| L1_TCH | int64 | counter | Level 1 total cache hits |
| L2_TCH | int64 | counter | Level 2 total cache hits |
| L3_TCH | int64 | counter | Level 3 total cache hits |
| L1_TCA | int64 | counter | Level 1 total cache accesses |
| L2_TCA | int64 | counter | Level 2 total cache accesses |
| L3_TCA | int64 | counter | Level 3 total cache accesses |
| L1_TCR | int64 | counter | Level 1 total cache reads |
| L2_TCR | int64 | counter | Level 2 total cache reads |
| L3_TCR | int64 | counter | Level 3 total cache reads |
| L1_TCW | int64 | counter | Level 1 total cache writes |
| L2_TCW | int64 | counter | Level 2 total cache writes |
| L3_TCW | int64 | counter | Level 3 total cache writes |

| **Mnemonics** | **Data Type** | **Unit** | **Descriptions** |
|---|---|---|---|
| FML_INS | int64 | counter | Floating point multiply instructions |
| FAD_INS | int64 | counter | Floating point add instructions |
| FDV_INS | int64 | counter | Floating point divide instructions |
| FSQ_INS | int64 | counter | Floating point square root instructions |
| FNV_INS | int64 | counter | Floating point inverse instructions |

Table C.2: Mnemonics for measured timing and counter metrics.

| **Mnemonics** | **Data type** | **Unit** | **Descriptions** |
|---|---|---|---|
| odata | double | microsecond | Data movement |
| odata_l21 | double | microsecond | L2 to L1 |
| odata_l23 | double | microsecond | L3 to L2 |
| odata_send | double | microsecond | SEND |
| odata_recv | double | microsecond | RECV |
| odata_p2p | double | microsecond | Point to Point communication |
| odata_col | double | microsecond | Collective communication |
| odata_put | double | microsecond | PUT |
| odata_get | double | microsecond | GET |
| odata_fread | double | microsecond | File System read |
| odata_fwrite | double | microsecond | File System write |
| odata_fother | double | microsecond | Other file system operation |
| osync | double | microsecond | Synchronization |
| osync_bar | double | microsecond | Barriers in Single address space |
| osync_lock | double | microsecond | Lock in single address space |
| osync_cond | double | microsecond | Conditional variable in single address space |
| osync_mpbar | double | microsecond | Barriers in multiple addresses space |
| osync_dcs | double | microsecond | Deferred communication synchronization |
| osync_crs | double | microsecond | Collective RMA synchronization |
| osync_rlo | double | microsecond | RMA Locks |
| octrp | double | microsecond | Control of parallelism |
| octrp_sched | double | microsecond | Schedule |
| octrp_insp | double | microsecond | Inspector |
| octrp_exec | double | microsecond | Executor |
| octrp_fkjn | double | microsecond | Fork/join threads |
| octrp_infl | double | microsecond | Initialize/Finalize message passing |
| octrp_sp | double | microsecond | Spawn processes |
| oadd | double | microsecond | Additional overhead |
| oadd_algr | double | microsecond | Overhead due to algorithm change |
| oadd_comp | double | microsecond | Overhead due to compiler changes |

| **Mnemonics** | **Data type** | **Unit** | **Descriptions** |
|---|---|---|---|
| oadd_dtc | double | microsecond | Overhead due to data type conversion |
| oadd_pui | double | microsecond | Overhead of processing unit information |
| olopa | double | microsecond | Overhead of loss parallelism |
| olopa_unpar | double | microsecond | Unparallelized code |
| olopa_repl | double | microsecond | Replicated code |
| olopa_ppar | double | microsecond | Partial parallelized code |
| oall_ident | double | microsecond | Identified Overhead |
| oall_unid | double | microsecond | Unidentified Overhead |
| oall | double | microsecond | All above overhead |

Table C.3: Mnemonics for overhead metrics.

# D

# SIS Command-line Options and APIs

This chapter outlines features of SCALEA Instrumentation System (SIS). More detailed information can be found [243].

## D.1 Command-line Options

SIS supplies various commands for controlling the instrumentation of parallel programs of many programming paradigms. The text bellow shows the commands that allow the user to specify programming models of input programs and to select the instrumentation library. To invoke SIS:

```
fsis [options] files...
```

with main options shown in Figure D.1.

In the following, we detail main options.

`-VH[`*`HPF+ Code regions`*`]` Specifies the input programs, which are HPF+ programs, and the HPF code regions to be instrumented.

`-VM[`*`MPI code regions`*`]` Specifies the input programs, which are MPI programs, and the MPI code regions to be instrumented.

`-VS[`*`OpenMP code regions`*`]` Specifies the input programs, which are OpenMP or sequential programs, and the OpenMP code regions to be selected. Note that the user has to specify option `-P5` to enable OpenMP directives; otherwise OpenMP directives will be omitted.

`-VLS` Specifies SISPROFILING instrumentation library.

`-C -O[`*`Output directory`*`]` Specifies the directory where the instrumented files are saved.

`-O[`*`outputfile`*`]` Specifies the output of instrumentation (instrumented program) to be saved in only one file named `outputfile`. This option is only valid for HPF programs.

By default, with a given input source file named `filename.[ext]`, the corresponding instrumented file will be `filename_sis.[ext]`.

```
     -F     free source form
     -P5    OMP directives allowed
     -P7    SIS directives allowed


     -C  code generation. With -C, -O<dirname> mode will be used.

     -Ofilename
            name of output file
            The file <filename> contains all the input files
     -Odirname
            name of output directory
            In this case for each inputfile,
            <dirname>/<input_filename>_sis.f90 will be generated.
            Useful for independent compilation.
            See "-C" for triggering between these two modes.
     -W0    no output of warnings
     -W[1]  output of warnings about possible error situations
     -W2    output of warnings about used extensions of
            Fortran 90
     -W4    output of warnings about used obsolescent features
            of Fortran 90

     -Ipathname
            add pathname to the list of directories in which to
            search for INCLUDE files or automatically imported
            modules.
     -VH[coderegions]    enable instrumentation of HPF programs
     -VS[coderegions]    enable instrumentation of OpenMP/sequential
                         programs
     -VM[coderegions]    enable instrumentation of MPI programs
     -VR[coderegions]    set generic code regions
     -VKn                set event counter n
     -VLS                use SISPROFILING Library
```

**Fig. D.1.** SIS main options

When instrumenting hybrid programs the user has to combine options for specific programming models. For examples, to instrument an OpenMP/MPI program the option `-p5 -p7 -VM[...] -VS[...]` is used.

Pre-defined code regions are classified into generic code regions and specific code regions. Table D.1 specifies the pre-defined common code region types. These types are set by using `-VR` options. For example, `-VRSLRW` means to instrument subroutines, loops and read/write statements. Note that the option `-VRL` (instrumentation of all loops) will override the option `-VRU` (instrumentation of outermost loops).

| Parameters | Generic code regions |
|---|---|
| P | Entire program |
| Y | Procedures |
| S | Subroutines calls |
| F | Function calls |
| L | All loops |
| U | Outermost loop |
| R | All read statements |
| W | All write statements |
| B | Arbitrary code regions |

**Table D.1.** Command for control common code regions

Table D.2 specifies pre-defined HPF code region types that are set by using `-VH` options. Table D.3 specifies pre-defined MPI code region types that are set by using `-VM` options. Table D.4 specifies pre-defined OpenMP code region types that are set by using `-VS` options.

| Parameters | HPF+ code regions |
|---|---|
| D | Work distribution |
| N | Inspector |
| X | Executor |
| G | Gather |
| T | Scatter |
| I | INDEPENDENT loops |
| R | All parallel read statement |
| W | All parallel write statement |
| E | All above code regions |

**Table D.2.** Parameters for controlling HPF+ code regions.

| Parameters | MPI Code regions |
|---|---|
| I | Initialization and Finalization |
| P | Point to point communication |
| C | Collective communication routines |
| O | MPI IO routines |
| B | Barrier routines |
| T | Data type routines |
| E | All MPI routines |

**Table D.3.** Parameters for controlling MPI code regions.

| Parameters | OMP Code regions |
|---|---|
| A | OMP PARALLEL |
| D | OMP PARALLEL DO |
| S | OMP PARALLEL SECTIONS |
| W | OMP PARALLEL WORSHARE |
| O | OMP DO |
| Z | OMP SECTIONS |
| K | OMP WORKSHARE |
| I | OMP SINGLE |
| M | OMP MASTER |
| B | OMP BARRIER |
| T | OMP ATOMIC |
| R | OMP ORDERED |
| F | OMP FLUSH |
| E | all above code regions |

**Table D.4.** Parameters for controlling OpenMP code regions.

## D.2 High-level Instrumentation APIs

SISHLLIB (SIS High-level Library) provides a full Fortran90 OpenMP/MPI /HPF frontend that allows external tools to instrument an abstract syntax tree at a very high-level with C-function calls and to generate source code.

`void sishl_insert_fstmt_acr(tTree tree, char *name );`
This routine is used to insert SIS directives that mark an arbitrary code region and its `name`. The region is determined by a single ATS tree node pointed by `tree`.

`void sishl_insert_acr(tTree beginTree, tTree endTree, char *name);`
This routine is used to insert SIS directives that mark an arbitrary code region and its `name`. The arbitrary code region is determined by the begin tree(`beginTree`) and the end (tree `endTree`).

`void sishl_insert_local_fstmt(tTree tree, char *cr_mnem[], int ncr_mnem, char * perf_mnem[], int nperf_mnem, char * name);`
This routine is used to insert SIS directives that enclose a set of code regions named `name`. Code regions in the selected set that are included in the list of `cr_mnem` will be measured with performance metrics given by the list of `perf_mnem`. The selected set of code regions is pointed by a single ATS tree node (`tree`).

`void sishl_insert_local(tTree beginTree,tTree endTree, char * cr_mnem[], int ncr_mnem, char * perf_mnem[], int nperf_mnem, char *name);`
This routine is used to insert SIS directives that enclose a set of code regions

named `name`. Code regions in the selected set that are included in the list of `cr_mnem` will be measured with performance metrics given by the list of `perf_mnem`. The selected set is determined by the begin tree (`beginTree`) and the end (tree `endTree`).

```
void sishl_insert_global(tTree tree, char * cr_mnem[],int ncr_mnem,
char * perf_mnem[], int nperf_mnem);
```
This routine is used to insert a global SIS directive with list of code regions (`cr_mnem`) and their performance metrics(`perf_mnem`); the directive affects in the entire program unit. The user can select any node in a program unit and insert the global directive. Since the global directive affects entire program unit, we don't need to insert it several times.

```
void sishl_measure_enable(tTree tree, int before);
```
This function will insert SIS MEASURE ENABLE directives before/after a statement determined by `tree`. If `before` is 1 then directive is inserted before the statement otherwise it is inserted after.

```
void sishl_measure_disable(tTree tree, int before);
```
This function will insert SIS MEASURE DISABLE directives before/after a statement determined by `tree`. If `before` is 1 then directive is inserted before the statement otherwise it is inserted after.

## D.3 On Detailing Instrumentation for OpenMP and MPI Code Regions

| **Original Source Code** | **Instrumented Source Code** |
|---|---|
| | sis_start($PB_1$, CR_OMPPA) |
| | sis_master_start($PB_2$, CR_OMPBPA) |
| !$OMP PARALLEL | !$OMP PARALLEL |
| | sis_master_stop( $PB_2$) |
| statements | statements |
| | sis_master_start($PB_3$,CR_OMPEPA) |
| !$OMP END PARALLEL | !$OMP END PARALLEL |
| | sis_master_stop( $PB_3$) |
| | sis_stop($PB_1$) |
| | sis_start( $PB_1$, CR_OMPDO) |
| !$OMP DO SCHEDULE | !$OMP DO SCHEDULE() |
| DO I=... | DO I=... |
| statements | statements |
| END DO | END DO |
| !$OMP END DO [NOWAIT] | !$ OMP END DO NOWAIT |
| | sis_stop($PB_1$) |

| **Original Source Code** | **Instrumented Source Code** |
|---|---|
| | [sis_start($PB_2$,CR_OMPBAR) |
| | !$ OMP BARRIER |
| | sis_stop($PB_2$)] |
| | sis_start( $PB_1$, CR_OMPSE) |
| !$OMP SECTIONS | !$OMP SECTIONS |
| !$OMP SECTION | !$OMP SECTION |
| | sis_start($PB_2$, CR_OMPISE) |
| statements | statements |
| | sis_stop($PB_2$) |
| !$OMP END SECTION | !$OMP END SECTION |
| ... | ... |
| !$OMP END SECTIONS | !$OMP END SECTIONS |
| | sis_stop($PB_1$) |
| | sis_start($PB_1$,CR_OMPSI) |
| !$OMP SINGLE | !$OMP SINGLE |
| | sis_start( $PB_2$,CR_OMPISI) |
| statements | statements |
| | sis_stop($PB_2$) |
| !$OMP END SINGLE [NOWAIT] | !$OMP END SINGLE [NOWAIT] |
| | sis_stop($PB_1$) |
| !$OMP MASTER | !$OMP MASTER |
| | sis_start($PB_1$,CR_OMPMA) |
| statements | statements |
| | sis_stop($PB_1$) |
| !$OMP END MASTER | !$OMP END MASTER |
| | sis_start( $PB_1$, CR_OMPCR) |
| !$OMP CRITICAL | !$OMP CRITICAL |
| | sis_start($PB_2$, CR_OMPICR) |
| statements | statements |
| | sis_stop($PB_2$) |
| !$OMP END CRITICAL | !$OMP END CRITICAL |
| | sis_stop($PB_2$) |
| | sis_start($PB_1$, CR_OMPBA) |
| !$OMP BARRIER | !$OMP BARRIER |
| | sis_stop($PB_1$) |
| | sis_start($PB_1$, CR_OMPOR) |
| !$OMP ORDERED | !$OMP ORDERED |
| | sis_start($PB_2$,CR_OMPIOR) |
| statements | statements |
| | sis_stop($PB_2$) |
| !$OMP END ORDERED | !$OMP END ORDERED |

| **Original Source Code** | **Instrumented Source Code** |
|---|---|
| | sis_stop($PB_1$) |
| !$OMP FLUSH | sis_start($PB_1$,CR_OMPFL)<br>!$OMP FLUSH<br>sis_stop($PB_1$) |
| call MPI_function_name() | sis_start($PB_{id}$,*cr_type*)<br>call MPI_function_name ()<br>sis_stop($PB_{id}$) |

Table D.5: Examples of instrumentation for OpenMP and MPI code regions implemented in SIS.

# E

# Performance Database

## E.1 Database Schema

The data schema of the experiment data repository is presented as follows.

```
CREATE TABLE PMC (
       pmcID     SERIAL PRIMARY KEY,
       pmcMetricName       TEXT,
       pmcMetricUnit       TEXT,
       pmcMetricDataType   TEXT,
       pmcMetricDesc       TEXT
);

CREATE TABLE VirtualMachine (
    virtualMachineID       SERIAL PRIMARY KEY,
    name                   TEXT
);

CREATE TABLE VirtualNode (
    virtualNodeID      SERIAL PRIMARY KEY,
    name               TEXT,
    numCpu        INTEGER,
    virtualMachineID  INTEGER REFERENCES VirtualMachine
);

CREATE TABLE Cluster (
    clusterID      SERIAL PRIMARY KEY,
    name             TEXT,
    totalFlops      REAL
);

CREATE TABLE Network (
    networkID          SERIAL PRIMARY KEY,
    name               TEXT,
    bandwidth          REAL,
    latency            REAL,
    clusterID          INTEGER REFERENCES Cluster
);

CREATE TABLE NetworkMPP2PPerf(
    colMPICommID     SERIAL PRIMARY KEY,
    mpilibName      TEXT,
    numNode          INTEGER,
    mpiBarrier       REAL,
    mpiBcast         REAL[5],
    networkID        INTEGER REFERENCES  Network
);


CREATE TABLE NetworkMPColPerf(
    perfMPICompNodeID       SERIAL PRIMARY KEY,
    mpilibName  TEXT,
    mpiSync          REAL[6],
    mpiAsync          REAL[6],
    mpiSsend          REAL[6],
    networkID            INTEGER REFERENCES Network
);

CREATE TABLE NodeSharedMemoryPerf(
    perfOMPCompNodeID    SERIAL PRIMARY KEY,
    omplibName  TEXT,
    clusterID            INTEGER REFERENCES Cluster
);

CREATE TABLE ComputationalNode (
    compNodeID       SERIAL PRIMARY KEY,
    hostName         TEXT,
    hostAliases      TEXT[],
    hostAddresses    TEXT[],
    systemModel      TEXT,
    physMem      TEXT,
    virtMem      TEXT,
    numCpu        INTEGER,
    cpuType        TEXT,
    cpuSpeed        TEXT,
    osName        TEXT,
    osVersion        TEXT,
    hardDiskSize       INTEGER,
    dataCacheSize      INTEGER[],
    cacheMissPenalty   INTEGER[],
    clusterID        INTEGER REFERENCES Cluster
);

CREATE TABLE Application (
    applicationID      SERIAL PRIMARY KEY,
    name               TEXT
);

CREATE TABLE Version (
    versionID          SERIAL PRIMARY KEY,
    versionInfo        TEXT ,
    applicationID      INTEGER NOT NULL REFERENCES Application
);

CREATE TABLE SourceFile (
    sourceFileID       SERIAL PRIMARY KEY,
    name               TEXT   NOT NULL,
    contents           TEXT,
    location        TEXT,
    versionID          INTEGER NOT NULL References Version
);

CREATE TABLE CodeRegion (
    codeRegionID       SERIAL PRIMARY KEY,
    startPos_x         INTEGER NOT NULL,
    startPos_y         INTEGER NOT NULL,
    endPos_x           INTEGER NOT NULL,
    endPos_y           INTEGER NOT NULL,
    CodeRegionType     TEXT,
    CodeRegionUnit     TEXT,
    codeRegion        INTEGER NOT NULL,
    sourceFileID       INTEGER,
    parentCodeRegionID INTEGER REFERENCES CodeRegion
);

CREATE TABLE Experiment (
    experimentID       SERIAL PRIMARY KEY,
    info       TEXT,
    startTime          TIMESTAMP,
    endTime            TIMESTAMP,
    commandLine        TEXT,
    compiler           TEXT,
    compilerOptions    TEXT,
    versionID          INTEGER NOT NULL REFERENCES Version
);

CREATE TABLE ProcessingUnit (
    puID                SERIAL PRIMARY KEY,
    computationalNode TEXT NOT NULL,
    processID INTEGER NOT NULL,
    threadID INTEGER NOT NULL
);

CREATE TABLE RegionSummary (
    regionSummaryID     SERIAL PRIMARY KEY,
    computationalNode TEXT NOT NULL,
    processID INTEGER NOT NULL,
    threadID INTEGER NOT NULL,
    codeRegion INTEGER ,
    puID                INTEGER REFERENCES ProcessingUnit,
    parentRegion INTEGER,
    codeRegionGroup INTEGER,
    numberCalls INTEGER,
    numberSubs INTEGER,
    codeRegionID INTEGER,
    regionSummaryIDParent    INTEGER,
    experimentID        INTEGER NOT NULL REFERENCES Experiment
);

CREATE TABLE TimingMetrics (
    name                TEXT NOT NULL,
    value               DOUBLE PRECISION NOT NULL,
    regionSummaryID     INTEGER NOT NULL REFERENCES RegionSummary
);

CREATE TABLE HardwareMetrics (
    name                TEXT NOT NULL,
    value               DOUBLE PRECISION NOT NULL,
    regionSummaryID     INTEGER NOT NULL REFERENCES RegionSummary
);

CREATE TABLE TemporalOverheadMetrics (
    name                TEXT NOT NULL,
    value               DOUBLE PRECISION NOT NULL,
    regionSummaryID     INTEGER NOT NULL REFERENCES RegionSummary
);
```

# F

# Fuzzy Logic

## F.1 Introduction

Fuzzy logic (FL) was introduced in the theory of fuzzy sets by Lotfi Zadeh [273, 274]. FL is a superset of Boolean logic dealing with the concept of partial truth. In classical logic, everything can be expressed in binary terms: 0 or 1 (*false* or *true*), FL extends Boolean truth values with degrees of truth.

Two main directions in FL are *fuzzy logic in the broad sense* and *fuzzy logic in the narrow sense* [276, 228]: The first direction covers mainly topics concerning vagueness such as fuzzy control, analysis of vagueness in natural language, etc. Soft computing techniques that we are using, e.g. computational methods tolerant to imprecision, fast and simple approximate solutions, fall into this direction. The latter is a part of symbolic logic, many-valued logic relevant for reasoning under vagueness.

In the following, we outline main concepts of FL. More detail of FL can be found in [228, 151, 280, 180].

## F.2 Fuzzy Sets and Membership Functions

### F.2.1 Fuzzy Sets

A fuzzy set is an extension of the classical set (crisp set) theory. In classical set, an element is either in a set (the membership is full) or not in a set (the membership is *no*). In other words, the membership of an element $x$ in set $A$ is described by a characteristic function $\mu_A(x)$ as follows

$$\mu_A(x) : A \rightarrow \{0, 1\} \tag{F.1}$$

where 1 represents full membership ($x \in A$) and 0 represents no membership ($x \notin A$).

Fuzzy set theory extends the membership concept by defining *graded degree of membership.* A fuzzy set is characterized by a membership function, which maps the members of the universe of discourse into the unit interval $[0, 1]$. The value 0 means that the member does not belong to the given set, 1 indicates a fully included member. The values between 0 and 1 indicates a graded degree of membership.

Let $X$ be the universe of discourse and its elements are denoted by x. A fuzzy set $A$ in $X$ is defined as a set of ordered pairs

$$A = \{(x, \mu_A(x))|x \in X\} \tag{F.2}$$

where $\mu_A(x)$ is called the membership function of $x$ in $A$. The membership function maps each element of $X$ to a membership value in the interval $[0, 1]$.

### F.2.2 Common Membership Functions

Several functions can be used as membership functions. Most common membership functions are given below.

- **triangular**: The triangular function of x depends on three parameters a, b, and c, and is given by

$$f(x; a, b, c) = max(min(\frac{x-a}{b-a}, \frac{c-x}{c-b}), 0) \tag{F.3}$$

- **trapezoid**: The trapezoidal function of x depends on four parameters a, b, c, and d, and is given by

$$f(x; a, b, c, d) = max(min(\frac{x-a}{b-a}, 1, \frac{d-x}{d-c}), 0) \tag{F.4}$$

- **generalized bell**: The generalized bell function of x depends on three parameters a, b, and c, and is given by

$$f(x; a, b, c) = \frac{1}{1 + |\frac{x-c}{a}|^{2b}} \tag{F.5}$$

- **sigmoid**: The sigmoid function of x depends on two parameters a and c, and is given by

$$f(x; a, c) = \frac{1}{1 + e^{-a(x-c)}} \tag{F.6}$$

- **S-function**: This function of x has S-shape and depends on two parameters a and b, and is given by

$$f(x; a, b) = \begin{cases} 0 & x \leq a, \\ 2(\frac{x-a}{b-a})^2 & a \leq x \leq \frac{a+b}{2} \\ 1 - 2(\frac{b-x}{b-a})^2 & \frac{a+b}{2} \leq x \leq b \\ 1 & x \geq b \end{cases} \tag{F.7}$$

- **Z-function**: This function of x has Z-shape and depends on two parameters a and b, and is given by

$$f(x;a,b) = \begin{cases} 1 & x \leq a, \\ 1 - 2(\frac{x-a}{b-a})^2 & a \leq x \leq \frac{a+b}{2} \\ 2(\frac{b-x}{b-a})^2 & \frac{a+b}{2} \leq x \leq b \\ 0 & x \geq b \end{cases} \tag{F.8}$$

- **Gaussian function**: The symmetric Gaussian function of x depends on two parameters $\sigma$ and c, and is given by

$$f(x;\sigma,c) = e^{\frac{-(x-c)^2}{2\sigma^2}} \tag{F.9}$$

## F.3 Basic Operations on Fuzzy Sets

Fuzzy set operations are analogous to classical set operations

**Definition F.1 (Fuzzy Set Intersection).** *The membership function $\mu_{A\cap B}(x)$ of the intersection $A \cap B$ is defined by*

$$\mu_{A\cap B}(x) = min\{\mu_A(x), \mu_B(x)\}, x \in X \tag{F.10}$$

**Definition F.2 (Fuzzy Set Union).** *The membership function $\mu_{A\cup B}(x)$ of the union $A \cup B$ is defined by*

$$\mu_{A\cup B}(x) = max\{\mu_A(x), \mu_B(x)\}, x \in X \tag{F.11}$$

**Definition F.3 (Fuzzy Set Complement).** *The membership function $\mu_{\bar{A}}(x)$ of the complement of a normalized fuzzy set $A, \mu_A(x)$ is defined by*

$$\mu_{\bar{A}}(x) = 1 - \mu_A(x), x \in X \tag{F.12}$$

## F.4 Fuzzy Modifiers (Hedges)

A fuzzy *modifier* (hedge) is an operation that modifies the meaning of a term representing a fuzzy set (in other words, a fuzzy modifier modifies a fuzzy set). For example, let *poor* be a fuzzy set, then we can apply modifiers such as *very, slightly* to *poor*, which result in new fuzzy sets such as *very poor, slight poor.* Terms such as *very, slightly* are called fuzzy modifiers. Theoretically, let $\mu_A(x)$ be the membership function of $x \in A$, the membership function $\mu_{modifier(A)}(x)$ is defined by

$$\mu_{modifier(A)}(x) = f(\mu_A(x)) \tag{F.13}$$

For example, the modifier *very* is defined as

$$\mu_{very(A)}(x) = (\mu_A(x))^2 \tag{F.14}$$

## F.5 Fuzzy Rules

Fuzzy rules are linguistic if-then constructions that contain fuzzy sets and fuzzy operators. An if-then rule statement is used to construct the fuzzy-based conditional statement. A fuzzy rule can be represented in the following form

**IF** *(x is A) AND (y is B) ... AND ...* **THEN** *consequent*

where $x, y$ are variables, $A, B$ are linguistic terms representing fuzzy sets. The part *(x is A) AND (y is B) ... AND ...* is called *antecedent* (or *premise*), and *consequent* is the conclusion of the rule. In FL, a rule is fired when there is a nonzero value of the antecedent.

Several rules constitute a fuzzy rule-based system.

# G

# Ontology and Ontology Languages

## G.1 Introduction

In this section, we briefly outline ontology and ontology languages. A greater detail of ontology and ontology languages can be found in [116, 90, 171, 111].

## G.2 An Overview of Ontology

Ontologies are a popular research topic in various communities such as representation and reasoning, information integration and cooperative information systems [90]. Ontologies were originally introduced by the artificial intelligence community to *"facilitate knowledge sharing and reuse"* [116, 90, 171] and recently ontology has been considered as the main tool that can be used to achieve the semantic interoperability in the Grid [72, 234]. A widely-accepted definition of ontology given by Tom Gruber is as follows.

**Definition G.1 (Ontology).** *"An ontology is a formal, explicit specification of a shared conceptualization". [116]*

This means that an ontology specifies common concepts and relationships that characterize a domain in an open and explicit form that are accepted by anyone in the domain. Thus these concepts and relationships can be shared. Typically, an ontology describes main concepts in a domain, the properties of each concept, and axioms that impose constraints on these concepts and properties. Ontologies may be categorized into different types of ontologies such as domain ontologies, task ontologies and application ontologies [90].

There are several reasons why ontology should be used [182]. One of key features of ontology is that it provides a shared and common understanding of some domain that can be used in the communication between people and application systems [116, 90]. For example, suppose many different tools, executed on different Grid sites, provide monitoring and measurement service, if these tools share and publish the same underlying ontology of descriptions, any performance analysis service can aggregate information from these sites in a unified way in order to conduct the analysis.

Moreover, because ontology describes concepts and relationships with a high level of expressiveness and detail, a set of ontology statements can allow us to infer another facts while that cannot be achieved with XML or relational database schema. We have to implement new functions including extra knowledge (e.g., for reasoning whether a property is an inverse of another property) in order to infer new facts from XML or relational data. Such knowledge (e.g., whether a property is an inverse of another property) are explicitly specified in the ontology, but not in XML or relational data. The ontology, like a treaty [18], will facilitate the performance data sharing.

## G.3 Ontology Languages

An ontology language [90, 171, 111] usually introduces (i) *concepts* (also known as classes, entities); concepts are used to describe general things in the domain of interest; (ii) *properties* of concepts (also known as slots, attributes, roles); properties are used to describe property values of those things; (iii) *relationships* between concepts (also known as associations), and (iv) additional *constraints* involving those things.

# References

1. Apart working group. http://www.kfa-juelich.de/apart/.
2. Cluster Analysis What is it?. http://149.170.199.144/multivar/ca.htm.
3. http://people.ee.ethz.ch/õetiker/webtools/rrdtool/.
4. http://www-unix.globus.org/toolkit/docs/3.2/core/developer/message_security.html.
5. http://www.hipersoft.rice.edu/open64/.
6. Intel VTune Analyzers. http://www.intel.com/software/products/vtune/.
7. LAM/MPI Parallel Computing. http://www.lam-mpi.org/.
8. *"NIST/SEMATECH e-Handbook of Statistical Methods", http://www.itl.nist.gov/div898/handbook/.* 31 March 2004.
9. Paradyn Parallel Performance Tools, http://www.cs.wisc.edu/paradyn/.
10. Postgresql 7.1.2. http://www.postgresql.org/docs/.
11. Sleepcat Berkeley DB, http://www.sleepycat.com.
12. The Scientific Processes, http://webpages.charter.net/kwingerden/erhs/aquarium/processs.htm.
13. The Workflow Management Coalition. http://www.wfmc.org.
14. Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl.
15. Whirl intermediate language specification. http://open64.sourceforge.net.
16. Worldflow Management Coalition: Terminology and glossary. technical report wfmc-tc-1011, feb 1999.
17. *Peer-to-Peer and Grids: Synergies and Opportunities, Workshop at GGF9*, Huron Room, Sheraton Chicago Hotel, October 5 2003. http://csag.ucsd.edu/P2P-Grid/.
18. Interview Tom Gruber. AIS SIGSEMIS Bulletin 1(3), October 2004. http://www.sigsemis.org/.
19. A. Cooke et al. R-GMA An Information Integration System for Grid Monitoring. In *Proceedings of 11th International Conference on Cooperative Information Systems (CoopIS 2003)*, Sicily,Italy, 3-7 November 2003.
20. W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.
21. Andrea F. Abate, Antonio Esposito, Nicola Grieco, and Giancarlo Nota. Workflow performance evaluation through wpql. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 489–495. ACM Press, 2002.

22. Dong H. Ahn and Jeffrey S. Vetter. Scalable Analysis Techniques for Microprocessor Performance Counter Metrics. In *IEEE/ACM SC'2002*, Baltimore, Maryland, November 2002.
23. Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, 40(8):102–114, Aug 2002.
24. Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Comput.*, 28(5):749–771, 2002.
25. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference*, pages 483–485, 1967.
26. Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 85–96, 1997.
27. ASKALON - A Programming Environment and Tool Set for Cluster and Grid Computing. http://dps.uibk.ac.at/projects/askalon. Distributed and Parallel Systems Group, Institute for Computer Science, University of Innsbruck.
28. AustrianGrid. http://www.austriangrid.at/.
29. R. A. AYDT. SDDF: The pablo self-describing data format. Tech. rep., Department of Computer Science, University of Illinois, April 1994.
30. B. Tierney et. al. A Grid Monitoring Architecture. Technical report, Performance Working Group, Grid Forum, January 2002. http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-16-2.pdf.
31. Mark Baker and Garry Smith. GridRM: An Extensible Resource Management System. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER'03)*, pages 207–215, Hong Kong, December 01-04 2003. IEEE Computer Society Press.
32. Z. Balaton and G. Gombas. Resource and Job Monitoring in the Grid. In *Proceedings. Euro-Par 2003 Parallel Processings*, Klagenfurt, Austria, 2003.
33. Zoltan Balaton, Peter Kacsuk, Norbert Podhorszki, and Ferenc Vajda. From Cluster Monitoring to Grid Monitoring Based on GRM. In *Proceedings. 7th EuroPar'2001 Parallel Processings*, pages 874–881, Manchester, UK, 2001.
34. Bartosz Balis, Marian Bubak, Włodzimierz Funika, Tomasz Szepieniec, and Roland Wismüller. An infrastructure for Grid application monitoring. *LNCS*, 2474:41–49, 2002.
35. M.K. Bane and G.D. Riley. Automatic overheads profilers for openmp codes. In *Second European Workshop on OpenMP proceedings(EWOMP 2000)*, Edinburgh, Scotland, September 2000.
36. Olof Barring. Towards automation of computing fabrics using tools from the fabric management workpackage of the eu datagrid project. *ECONF*, C0303241:MODT004, 2003.
37. S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming, IOS Press, The Netherlands*, 7(1):67–81, 1999.
38. S. Benkner and T. Brandes. High-Level Data Mapping for Clusters of SMPs. In *Proceedings 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, San Francisco, USA, April 2001. Springer-Verlag. Lecture Notes in Computer Science, Vol. 2026.

39. Siegfried Benkner. HPF+: High Performance Fortran for advanced scientific and engineering applications. *Future Generation Computer Systems*, 15(3):381–391, ???? 1999.
40. Siegfried Benkner, Erwin Laure, and Hans Zima. HPF+: An Extension of HPF for Advanced Industrial Applications. Technical Report TR 99-1, Institute for Software Technology and Parallel Systems, University of Vienna, January 1999.
41. Fran Berman, Geoffrey Fox, Anthony J. G. Hey, and Tony Hey. *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons, Inc., 2003.
42. J. C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms.* Plenum Press, New York, 1981.
43. Dileep Bhandarkar and Richard Brunner. VAX vector architecture. In *Proc. 17th Annual Symposium on Computer Architecture (17th ISCA'90), Computer Architecture News*, pages 204–215. ACM, June 1990. Published as Proc. 17th Annual Symposium on Computer Architecture (17th ISCA'90), Computer Architecture News, volume 18, number 2.
44. J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao. ABLE: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002.
45. BISC: The Berkeley Initiative in Soft Computing. http://www-bisc.cs.berkeley.edu/.
46. P. Blaha, K. Schwarz, and J. Luitz. WIEN97, Full-potential, linearized augmented plane wave package for calculating crystal properties. Institute of Technical Electrochemistry, Vienna University of Technology, Vienna, Austria, ISBN 3-9501031-0-4, 1999.
47. OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface 2.0.* OpenMP Organisation, December 2000. Also avaible at http://www.openmp.org.
48. P. Bosc and O. Pivert. Fuzzy queries and relational databases. In *Proceedings of the 1994 ACM symposium on Applied computing*, pages 170–174. ACM Press, 1994.
49. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings SC'2000*, November 2000.
50. Marian Bubak, Włodzimierz Funika, and Roland Wismüller. The CrossGrid performance analysis tool for interactive Grid applications. *LNCS*, 2474:50–60, 2002.
51. Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
52. J.M. Bull. A Hierarchical classification of Overheads in Parallel Programs. In P. Croll I. Jelly, I. Gorton, editor, *Proceedings of Firs IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 208–219. Chapman Hall, March 1996.
53. Rajkumar Buyya, editor. *High Performance Cluster Computing, Volume 1: Architecture and Systems.* Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 1999.
54. C. Lee et. al. A grid programming primer. Technical report, Advanced Programming Models Research Group, Grid Forum, August 2001. http://www.eece.unm.edu/ãpm/docs/APM_Primer_0801.pdf.

55. Harold W. Cain, Barton P. Miller, and Brian J.N. Wylie. A Callgraph-Based Search Strategy for Automated Performance Diagnosis. In *Euro-Par 2000 Parallel Processing*, pages 108–122, 2000.
56. Maria Calzarossa, Luisa Massari, and Daniele Tessera. A methodology towards automatic performance analysis of parallel applications. *Parallel Comput.*, 30(2):211–223, 2004.
57. Franck Cappello and Daniel Etiemble. MPI versus MPI+openMP on IBM SP for the NAS benchmarks. In *Proceedings of Supercomputing'2000 (CD-ROM)*, Dallas, TX, November 2000. IEEE and ACM SIGARCH. LRI.
58. Jorge Cardoso, Amit P. Sheth, and John Miller. Workflow quality of service. In *Proceedings of the IFIP TC5/WG5.12 International Conference on Enterprise Integration and Modeling Technique*, pages 303–311. Kluwer, B.V., 2003.
59. Jordi Caubet, Judit Gimenez, Jesus Labarta, and Luiz DeRose. A dynamic tracing mechanism for performance analysis of OpenMP applications. *LNCS*, 2104:53–??, 2001.
60. IBM T.J. Watson Research Center. ABLE Rule Language: User's Guide and Reference. Version 2.1.0.
61. Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Memon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.
62. Ann Chervenak, Carl Kesselman, Charles Salisbury, Ian Foster, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets, July 28 2000.
63. Madhu Chetty and Rajkumar Buyya. Weaving computational grids: How analogous are they with electrical grids? *Computing in Science and Engg.*, 4(4):61–71, 2002.
64. George Chin, Karen Schuchardt, James Myers, and Debbie Gracio. Participatory Workflow Analysis: Unveiling Scientific Research Processes with Scientists. In *Proceedings of the Sixth Biennial Participatory Design Conference, Nov. 28-Dec. 1, 2000, NY, NY.*
65. P. Cicotti, Michela Taufer, and Andrew A. Chien. DGMonitor: A Performance Monitoring Tool for Sandbox-Based Desktop Grid Platforms. In *Third International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO), CD-ROM / Abstracts Proceedings, 26-30 April 2004,Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004.
66. Common Information Model (CIM). http://www.dmtf.org/standards/standard_cim.php.
67. Earl Cox. Fuzzy sql: A tool for finding the truth. the power of approximate database queries. 1999. Scianta Intelligence. http://scianta.com/pubs/AR-PA-008.htm.
68. Mark E. Crovella and Thomas J. LeBlanc. Parallel performance prediction using lost cycles analysis. In IEEE, editor, *Proceedings, Supercomputing '94: Washington, DC, November 14–18, 1994*, Supercomputing, pages 600–609, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.
69. David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc., 1997.
70. K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th*

*IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

71. DAML+OIL. http://www.daml.org/2001/03/daml+oil-index.html.
72. D. De Roure, N.R. Jennings, and N.R. Shadbolt. The semantic grid: A future e-science infrastructure. In F. Berman, G. Fox, and A. J. G. Hey, editors, *Grid Computing - Making the Global Infrastructure a Reality*, pages 437–470. John Wiley and Sons Ltd., 2003.
73. J.E. Lopez de Vergara, V.A. Villagra, J.I. Asensio, and J. Berrocal. Ontologies: Giving semantics to network management models. *IEEE Network*, 17(3):15–21, May-June 2003.
74. Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–39, 2003.
75. L. DeRose, T. Hoover Jr., and J. Hollingsworth. The dynamic probe class library: An infrastucture for developing instrumentation for performance tools. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS-01)*, pages 66–66, Los Alamitos, CA, April 23–27 2001. IEEE Computer Society.
76. U. Detert. Vector processing on CRAY-1 and CRAY X-MP. In Jozef T. Devreese and Piet Van Camp, editors, *Supercomputers in Theoretical and Experimental Science*, pages 13–31. Plenum Press, New York, 1985.
77. Discovery and Monitoring Event Description (DAMED) Working Group. http://www-didc.lbl.gov/damed/.
78. E. Dockner and H. Moritsch. Pricing Constant Maturity Floaters with Embeeded Options Using Monte Carlo Simulation. Technical Report AuR_99-04, AURORA Technical Reports, University of Vienna, January 1999.
79. Robert Elfwing, Ulf Paulsson, and Lars Lundberg. Performance of SOAP in Web Service Environment Compared to CORBA. In *Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*, pages 84–, Gold Coast, Australia, December 04 - 06. IEEE Computer Society.
80. European Center for Parallelism of Barcelona. *MPItrace tool version 1.1, Instrumentation of generic MPI Applications, User's Guide.* Technical University of Catalonia, November 2000. Also available at http://www.cepba.upc.es/paraver/.
81. European Center for Parallelism of Barcelona. *OMPItrace tool version 1.1, Instrumentation of combined OpenMP and MPI Applications, User's Guide.* Technical University of Catalonia, November 2000. Also available at http://www.cepba.upc.es/paraver/.
82. F. Wolf and B. Mohr. Automatic Performance Analysis of SMP Cluster Applications. Technical Report Tech. Rep. IB 2001-05, Research Center Juelich, 2001.
83. T. Fahringer, M. Gerndt, Bernd Mohr, Felix Wolf, G. Riley, and J. Träff. Knowledge Specification for Automatic Performance Analysis. Technical report, APART Working group, August 2001.
84. T. Fahringer, N. Mazzocca, M. Rak, S. Pllana, U. Villano, and G. Madsen. Performance Modeling of Scientific Applications: Scalability Analysis of LAPW0. In *11th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2003)*, Genoa, Italy, February 2003. IEEE Computer Society.

85. T. Fahringer, S. Pllana, and J. Testori. Teuta: Tool Support for Performance Modeling of Distributed and Parallel Applications. In *International Conference on Computational Science. Tools for Program Development and Analysis in Computational Science.*, Krakow, Poland, June 2004. Springer-Verlag.
86. T. Fahringer and C. Seragiotto. Automatic search for performance problems in parallel and distributed programs by using multi-experiment analysis. In *International Conference On High Performance Computing (HiPC 2002)*, Bangalore, India, December 2002. Springer Verlag.
87. T. Fahringer and C. Seragiotto. Aksum: A performance analysis tool for parallel and distributed applications. *Performance Analysis and Grid Computing*, October 2003.
88. Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto Junior, and Hong-Linh Truong. ASKALON: A Tool Set for Cluster and Grid Computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):143–169, February - April 2005.
89. Jay Fenlason and Richard Stallman. *GNU gprof*. Free Software Foundation, Inc., September 1997.
90. D. Fensel. *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag, Berlin, 2001.
91. Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.
92. International Organization for Standardization. *ISO/IEC 1539-1:1997: Information technology — Programming languages — Fortran — Part 1: Base language*. International Organization for Standardization, Geneva, Switzerland, 1997. ISO/IEC JTC 1/SC 22/WG 5. This is the Fortran 95 Standard. Available in English only.
93. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, pages 37–46, June 2002.
94. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. January 2002.
95. I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.
96. Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley Publishing Company, Reading, MA, 1995. Hardcover.
97. Ian Foster. What is a Grid? A Three Point Checklist, July 2002. http://www-fp.mcs.anl.gov/˜foster/Articles/WhatIsTheGrid.pdf.
98. Ian Foster and Adriana Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, February 2003.
99. Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, 1999.
100. Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS-98)*, pages 83–92, New York, November 3–5 1998. ACM Press.
101. FuzzyJ Toolkit. http://ai.iit.nrc.ca/IR_public/fuzzy/fuzzyJToolkit.html, 2004.
102. M. Geissler. *Interaction of High Intensity Ultrashort Laser Pulses with Plasmas*. PhD thesis, Vienna University of Technology, 2001.

103. M. Gerndt, A. Schmidt, M. Schulz, and R. Wismueller. Performance Analysis for Teraflop Computers - A Distributed Automatic Approach. In *Proceedings of 10th Euromicro Workshop on Parallel, Distributed, and Network-based Processing (EUROMICRO-PDP 2002)*, Canary Islands, SPAIN, 2002.
104. M. Gerndt, R. Wismueller, Z. Balaton, G. Gombas, P. Kacsuk, Z. Nemeth, N. Podhorszki, H.-L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef. *Performance Tools for the Grid: State of the Art and Future*, volume 30 of *Research Report Series, Lehrstuhl fuer Rechnertechnik und Rechnerorganisation (LRR-TUM) Technische Universitaet Muenchen*. Shaker Verlag, 2004. ISBN 3-8322-2413-0.
105. Michael Gerndt and Bernd Mohr. Automatic Performance Analysis Roadmap Report. APART Deliverable, http://www.kfa-juelich.de/apart/result.html, January 2001.
106. GGF Network Measurements Working Group. http://forge.gridforum.org/projects/nm-wg/.
107. Global Grid Forum. http://www.gridforum.org/.
108. Globus Project. http://www.globus.org.
109. Pallas GmbH. VAMPIR: Visualization and Analysis of MPI Programs. http://www.pallas.com/e/products/vampir/index.htm.
110. Carole Goble and Bertram Ludaescher. Scientific Workflow Requirements, e-science workflow services,, 2003. http://www.nesc.ac.uk/esi/events/303/index.html.
111. Asuncion Gomez-Perez and Oscar Corcho. Ontology Languages for the Semantic Web. *IEEE Intelligent Systems*, 17(1):54–60, Jan/Feb 2002.
112. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. `gprof`: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*
113. Grid computing: What are the key components? http://www-106.ibm.com/developerworks/grid/library/gr-overview/.
114. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
115. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface.* Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, November 1999.
116. T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
117. gSOAP: C/C++ Web Services and Clients. http://www.cs.fsu.edu/˜engelen/soap.html.
118. D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. Netlogger: A toolkit for distributed system performance analysis. In *Proceedings of the IEEE Mascots 2000 Conference*, August 2000.
119. Herbert Haas, Michael Geissler, Hong-Linh Truong, and Armin Scrinzi. Communication Analysis of the RPP3D Code. Technical Report AURORATR2004-20, Institute for Software Science, University of Vienna, December 2004.
120. Robert J. Hall. Call Path Refinement Profiles. *IEEE Transactions on Software Engineering*, 21(6):481–496, June 1995.
121. Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, August 2000.

122. Matt Haynos. Perspectives on grid: Grid computing – next-generation distributed computing, 27 Jan 2004. http://www-106.ibm.com/developerworks/grid/library/gr-heritage/.
123. Yun He and Chris H. Q. Ding. MPI and openMP paradigms on cluster of SMP architectures: the vacancy tracking algorithm for multi-dimensional array transposition. In *SC'2002 Conference CD*, Baltimore, MD, November 2002. IEEE/ACM SIGARCH. pap325.
124. R. Hempel. The MPI standard for message passing. *LNCS*, 797:247–252, 1994.
125. D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of Supercomputing'2000 (CD-ROM)*, Dallas, TX, November 2000. IEEE and ACM SIGARCH. Edinburgh Parallel Computing Centre.
126. Hewlett Packard. *CXperf User's Guide*, June 1998.
127. High Performance Fortran Forum. High Performance Fortran Language Specification: Version 2.0. Technical report, Rice University, Houston,TX, January 1997.
128. Frank Hoeppner, Frank Klawonn, Rudolf Kruse, and Thomas Runkler. *Fuzzy Cluster Analysis*. Wiley, 1999. ISBN 0-471-98864-2.
129. Wolfgang Hoschek. *Grid Computing: Making the Global Infrastructure a Reality*, chapter Peer-to-Peer Grid Databases for Web Service Discovery, pages 491–539. Wiley Press, 2003.
130. HP Labs Semantic Web Research. http://www.hpl.hp.com/semweb/.
131. HPC Info. http://www.epcc.ed.ac.uk/HPCinfo/index.html.
132. Michael N. Huhns and Munindar P. Singh. "service-oriented computing: Key concepts and principles". *IEEE Internet Computing*, 9(1):75–81, 2005.
133. Junseok Hwang and Praveen Aravamudham. Middleware Services for P2P Computing in Wireless Grid Networks. *IEEE Internet Computing*, 8(4), 2004.
134. Iperf. http://dast.nlanr.net/projects/iperf/.
135. Bart Jacob, Luis Ferreira, Norbert Bieberstein, Candice Gilzean, Jean-Yves Girard, Roman Strachowski, and Seong (Steve) Yu. *Enabling Applications for Grid Computing with Globus*. IBM, June 18 2003. IBM Redbooks.
136. A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.
137. R. Jain. *The art of computer system performance analysis: techniques for experimental design, measurement, simulation and modeling*. John Wiley and Sons, Inc., New York USA, 1991.
138. Jena - A Semantic Web Framework for Java. http://jena.sourceforge.net.
139. JFreeChart. http://www.jfree.org/jfreechart/.
140. JGraph. http://www.jgraph.com/.
141. Joshy Joseph, Mark Ernest, and Craig Fellenstein. Evolution of grid computing architecture and grid adoption models. *IBM Systems Journal*, 43(4):624–645, 2004.
142. P. Kacsuk, G. Dozsa, J. Kovacs, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombas. P-GRADE: a Grid Programming Environment. *Journal of Grid Computing*, 1(2):171–197, 2003.
143. Karen Karavanic and Barton Miller. Improving online performance diagnosis by the use of historical performance data. In *Proceedings of Supercomputing'99 (CD-ROM)*, Portland, OR, November 1999. ACM SIGARCH and IEEE. University of Wisconsin.

144. Karen L. Karavanic and Barton P. Miller. Experiment management support for performance tuning. In ACM, editor, *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. ACM Press and IEEE Computer Society Press.
145. Karen L. Karavanic and Barton P. Miller. Experiment Management Support for Performance Tuning. In *Proceedings of Supercomputing'97 (CD-ROM)*, San Jose, CA, November 1997. ACM SIGARCH and IEEE.
146. Jeffrey O. Kephart and David M. Chess. Cover feature: The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.
147. Carl Kesselman, Ian Foster, and Steven Tuecke. The anatomy of the grid - enabling scalable virtual organizations, May 10 2001.
148. Kwang-Hoon Kim and Clarence A. Ellis. Performance analytic models and analyses for workflow architectures. *Information Systems Frontiers*, 3(3):339–355, 2001.
149. S. W. Kim and R. Eigenmann. Where does the speedup go: Quantitative modeling of performance losses in shared-memory programs. *Parallel Processing Letters*, 10(2/3):227–??, September 2000.
150. Thomas Kistler and Michael Franz. Computing the similarity of profiling data—heuristics for guiding adaptive compilation. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.
151. George J. Klir and Tina A. Folger. *Fuzzy sets, uncertainty, and information.* Prentice-Hall, Inc., 1987.
152. Christoforos Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Technical Report CSD-99-1059, University of California, Berkeley, August 27, 1999.
153. Christoforos Kozyrakis, Joseph Gebis, David Martin, Samuel Williams, Ioannis Mavroidis, Steven Pope, Darren Jones, and David Patterson. Vector IRAM: A media-enhanced vector processor with embedded DRAM. In IEEE, editor, *Hot Chips 12: Stanford University, Stanford, California, August 13–15, 2000*, pages ??–??, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. IEEE Computer Society Press.
154. Heather Kreger. Web Services Conceptual Architecture (WSCA 1.0). Prepared for Sun Microsystems, Inc., IBM Software Group, May 2001. http://www-4.ibm.com/software/solutions/webservices/pdf/WSCA.pdf+.
155. Sriram Krishnan, Patrick Wagstrom, and Gregor von Laszewski. GSFL : A Workflow Framework for Grid Services. Technical report, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, U.S.A., July 2002.
156. Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing:design and analysis of parallel algorithms.* Benjamin/Cummings, 1994.
157. Sami Lais. Grid computing. ComputerWorld, December 23 2002. http://www.computerworld.com/networkingtopics/networking/management/story/0,10801,76946,00.html.
158. DAML: The DARPA Agent Markup Language. http://www.daml.org/.
159. RDQL: RDF Data Query Language. http://www.hpl.hp.com/semweb/rdql.htm.
160. G. Laszewski, I. Foster, J. Gawor, and P. Lane. A java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(643-662), 2001.

161. OIL: Ontology Inference Layer. http://www.ontoknowledge.org/oil/.
162. Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27(10):15–26, Oct 1994.
163. Kathleen A. Lindlan, Janice E. Cuny, Allen D. Malony, Sameer Shende, Bernd Mohr, Reid Rivenburgh, and Craig Rasmussen. A tool framework for static and dynamic analysis of object-oriented software with templates. pages 49–49, 2000.
164. B. Ludaescher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*, 2005.
165. T. Ludwig and R. Wismueller. OMIS 2.0 — A universal interface for monitoring systems. *LNCS*, 1332:267–276, 1997.
166. A. D. Malony, S. Shende, R. Bell, K. Li, L. Li, and N. Trebon. Advances in the tau performance system. In *PADC*. Kluwer, 2003.
167. Allen Malony and Sameer Shende. Performance technology for complex parallel and distributed systems. In *In G. Kotsis and P. Kacsuk (Eds.), Third International Austrian/Hungarian Workshop on Distributed and Parallel Systems (DAPSYS 2000)*, pages 37–46. Kluwer Academic Publishers, Sept. 2000.
168. Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, May 2004.
169. Deborah L. McGuinness, Richard Fikes, James Rice, and Steve Wilder. An Environment for Merging and Testing Large Ontologies. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, Breckenridge, Colorado, USA, April 2000.
170. John Mellor-Crummey, Robert J. Fowler, Gabriel Marin, and Nathan Tallent. Hpcview: A tool for top-down analysis of node performance. *J. Supercomput.*, 23(1):81–104, 2002.
171. Leo J. Obrst Michael C. Daconta and Kevin T. Smith. *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management.* John Wiley & Sons, 2003.
172. B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, November 1995.
173. D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-To-Peer Computing. Technical Report HPL-2002-57, HP Labs, March 2002.
174. Tom M. Mitchell. *Machine Learning.* McGraw Hill, New York, US, 1997.
175. Bernd Mohr, Allen Malony, Sameer Shende, and Felix Wolf. Towards a performance tool interface for openmp: An approach based on directive rewriting. In *EWOMP'01 Third European Workshop on OpenMPI*, Sept. 2001.
176. Montage. http://montage.ipac.caltech.edu.
177. Douglas C. Montgomery. *Design and Analysis of Experiments.* John Wiley & Sons, fifth edition, 2003.
178. N. Mukherjee, G.D. Riley, and J.R. Gurd. FINESSE: A Prototype Feedback-Guided Performance Enhancement System. In *Proceedings of the 8th Euromicro on Conference Parallel and Distributed Processing*, pages 101–109. Rhodes, Greece, IEEE Computer Society Press, February 19 - 21 2000.

179. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, January 1996.
180. H. T. Nguyen and E. A. Walker. *A first Course in Fuzzy Logic.* Chapman & Hall/Crc, 2000.
181. J. O. Nicholson. The risc system/6000 smp system. In *Proceedings of the 40th IEEE Computer Society International Conference*, page 102. IEEE Computer Society, 1995.
182. N. Noy and D. L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. Technical Report KSL-01-05, Knowledge Systems Laboratory, March 2001.
183. N. F. Noy and M. A. Musen. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In *Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, Austin, TX, USA, 2000.
184. M. T. Oezsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Upper Saddle River, 2 edition, 1999.
185. B. R. Olszewski and J.-J. Guillemaud. The performance and performance methodology for a powerpc smp system. In *Proceedings of the 40th IEEE Computer Society International Conference*, page 116. IEEE Computer Society, 1995.
186. IEEE Task Force on Cluster Computing. http://www.ieeetfcc.org/.
187. R. A. Orchard. *FuzzyCLIPS V6.04 User's Guide.* KS Lab. Institute for Information Technology, NRC Canada, 1998.
188. OWL Web Ontology Language Reference. http://www.w3.org/tr/owl-ref/.
189. Home page of T.H. Kaiser. http://www.sdsc.edu/ tkaiser/.
190. M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing.
191. Insung Park, Michael Voss, Brian Armstrong, and Rudolf Eigenmann. Parallel programming and performance evaluation with the URSA tool family. *International Journal of Parallel Programming*, 26(5):541–??, ???? 1998.
192. V. Paxson. Towards a framework for defining internet performance metrics, 1996.
193. P. Pereira, Laurent Heutte, and Yves Lecourtier. Source-to-source instrumentation for the optimization of an automatic reading system. *The Journal of Supercomputing*, 18(1):89–104, 2001.
194. Gregory F. Pfister. *In search of clusters: The Ongoing Battle in Lowly Parallel Computing, 2nd ed.* Prentice Hall, Englewood Cliffs, NJ, 1995 edition, 1998. :1998 edition: ISBN 0-13-899709-8.
195. S. Pllana and T. Fahringer. UML Based Modeling of Performance Oriented Parallel and Distributed Applications. In *Proceedings of the 2002 Winter Simulation Conference*, San Diego, California, USA, December 2002. IEEE.
196. N. Podhorszki and P. Kacsuk. Design and implementation of a distributed monitor for semi-on-line monitoring of visualmp applications. In *DAPSYS'2000 Distributed and Parallel System, From Instruction Parallelism to Cluster Computing*, pages 23–32, Balatonfured,Hungary, 2000.
197. N. Podhorszki and P. Kacsuk. Monitoring Message Passing Applications in the Grid with GRM and R-GMA. In *Proceedings of EuroPVM/MPI'2003*, Venice, Italy, 2003.
198. Jerry L. Potter. *The Massively Parallel Processor.* MIT Press, 1985.
199. Power Grid Analogy. http://gridcafe.web.cern.ch/gridcafe/whatisgrid/dream/powergrid.html.

200. Radu Prodan. *Experiment Management, Performance Optimisation, and Tool Integration in Grid Computing.* PhD thesis, Vienna Technical University, October 2004.
201. Radu Prodan and Thomas Fahringer. ZENTURIO: An Experiment Management System for Cluster and Grid Computing. In *Proceedings of the 4th International Conference on Cluster Computing (CLUSTER 2002)*, Chicago, USA, September 2002. IEEE Computer Society Press.
202. Jaroslav Ramik. Soft computing - Overview and Recent Developments in Fuzzy Optimization. Technical report, Institute for Research and Applications of Fuzzy Modeling, University of Ostrava, 2001. Available as pdf at http://ac030.osu.cz/irafm/ps/softco01.pdf.
203. IBM Redbooks. *Introduction to Grid Computing with Globus.* IBM, October 1 2003.
204. D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.
205. Resource Description Framework (RDF). http://www.w3.org/rdf/.
206. Rita A. Ribeiro and Ana M. Moreira. Fuzzy query interface for a business database. *Int. J. Hum.-Comput. Stud.*, 58(4):363–391, 2003.
207. R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive control of distributed applications. In *Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, page 172. IEEE Computer Society, 1998.
208. Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using etch. In USENIX, editor, *The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, pages 1–7, Berkeley, CA, USA, August 1997. USENIX.
209. D. D. Roure, M. A. Baker, N. R. Jennings, and N. R. Shadbolt. *Grid Computing: Making the Global Infrastructure a Reality*, chapter "The Evolution of the Grid", pages 65–100. Wiley & Sons, 2003.
210. Richard M. Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.
211. Nayda G. Santiago, Diane T. Rover, and Domingo Rodriguez. A Statistical Approach for the Analysis of the Relation Between Low-Level Performance Information, the Code, and the Environment. In *Proceedings of 2002 International Conference on Parallel Processing Workshops (ICPPW'02)*, pages 282–, Vancouver, B.C., Canada, August 18-21 2002. IEEE Computer Society Press.
212. Bastin Tony Roy Savarimuthu, Maryam Purvis, and Martin Fleurke. Monitoring and controlling of a multi-agent based workflow system. In *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 127–132. Australian Computer Society, Inc., 2004.
213. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/tr/rdf-schema/, January 2003. W3C Working Draft.
214. G. Schuster, H. Neumann, H. Stuckenschmidt, H. Wache, S. H Ubner, T. V Ogele, and U. Visser. Ontology-based integration of information - A survey of existing approaches. February 16 2001.

215. Bradley W. Schwartz, Daniel A. Reed, Shields Shields, Luis F. Tavera, Phillip C. Roth, Roger J. Noe, and Ruth A. Aydt. Scalable performance analysis: The pablo performance analysis environment, January 06 1998.
216. Andy Seaborne. RDQL - A Query Language for RDF, 9 January 2004. W3C Member Submission. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.
217. C. Seragiotto, T. Fahringer, M. Geissler, G. Madsen, and H. Moritsch. On using aksum for semi-automatically searching of performance problems in parallel and distributed programs. In *11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP 2003)*, Genoa, Italy, February 2003. IEEE Computer Society Press.
218. Clovis Seragiotto, Hong-Linh Truong, Thomas Fahringer, Bernd Mohr, Michael Gerndt, and Tianchao Li. Monitoring and Instrumentation Requests for Fortran, Java, C and C++ Programs. Technical Report AURORATR2004-17, Institute for Software Science, University of Vienna, October 2004.
219. Clovis Seragiotto, Hong-Linh Truong, Thomas Fahringer, Bernd Mohr, Michael Gerndt, and Tianchao Li. Standardized Intermediate Representation for Fortran, Java, C and C++ Programs. Technical report, Institute for Software Science, University of Vienna, October 2004.
220. Albert Serra, Nacho Navarro, and Toni Cortes. DITools: Application-level support for dynamic extension and flexible composition. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 225–238, Berkeley, CA, June 18–23 2000. USENIX Ass.
221. T. Sheehan, A. Malony, and S. Shende. A runtime monitoring framework for tau profiling system. In *Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments(ISCOPE's 99)*, San Franciso, December 1999.
222. Sameer Shende. *The Role of Instrumentation and Mapping in Performance Measurement*. PhD thesis, University of Oregon, August 2001.
223. Munindar P. Singh and Mladen A. Vouk. Scientific workflows. In *Position paper in Reference Papers of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions*, May 1996.
224. Lorna Smith and Mark Bull. Development of mixed mode MPI/OpenMP applications. *Scientific Programming*, 9(2–3):83–98, Spring–Summer 2001.
225. Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1996.
226. Fengguang Song, Felix Wolf, Nikhil Bhatia, Jack Dongarra, and Shirley Moore. An algebra for cross-experiment performance analysis. In *ICPP*, pages 63–72. IEEE Computer Society, 2004.
227. John F. Sowa. *Knowledge Representation: logical, philosophical, and compuational foundations*. Brooks/Cole, Pacific Grove, CA, 2000.
228. Stanford Encyclopedia of Philosophy. Fuzzy Logic. http://plato.stanford.edu/entries/logic-fuzzy/.
229. Mehmet F. Su, Ihab El-Kady, David A. Bader, and Shawn-Yu Lin. A novel fdtd application featuring openmp-mpi hybrid parallelization. In *33rd International Conference on Parallel Processing (ICPP 2004), 15-18 August 2004, Montreal, Quebec, Canada*, pages 373–379. IEEE Computer Society, 2004.

230. Gescher system. http://gescher.vcpc.univie.ac.at.
231. Domenico Talia and Paolo Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7(04):96–95, 2003.
232. Domenico Talia and Paolo Trunfio. Web Services for Peer-to-Peer Resource Discovery on the Grid. In *DELOS Workshop: Digital Library Architectures*, S. Margherita di Pula, Cagliari, Italy, 24-25, June 2004. Edizioni Libreria Progetto, Padova.
233. H. Tangmunarunkit, S. Decker, and C. Kesselman. Ontology-based Resource Matching in the Grid—The Grid meets the Semantic Web. In *Proceedings of the Second International Semantic Web Conference*, Sanibel-Captiva Islands, Florida, USA, October 2003.
234. F. Tao, L. Chen, N. R. Shadbolt, G. Pound, and S. J. Cox. Towards the semantic grid: Putting knowledge to work in design optimisation. In *3rd International Conference on Knowledge Management (I-KNOW '03)*, pages 555–566, 2003.
235. V. Taylor, X. Wu, J. Geisler, X. Li, Z. Lan, R. Stevens, M. Hereld, and Ivan R.Judson. Prophesy:An Infrastructure for Analyzing and Modeling the Performance of Parallel and Distributed Applications. In *Proc. of HPDC's 2000*, Pittsburgh, August 2000. IEEE Computer Society Press.
236. The Condor Team. Dagman (directed acyclic graph manager). http://www.cs.wisc.edu/condor/dagman/.
237. The Grid Laboratory Uniform Environment (GLUE). http://www.cnaf.infn.it/ sergio/datatag/glue/index.htm.
238. The Three Types of Grids. http://nz.sun.com/2002-0708/grid/types.html.
239. The World Wide Web Consortium (W3C). http://www.w3.org/.
240. B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *Proceedings of IEEE High Performance Distributed Computing conference*, July 1998.
241. Brian Tierney, Brian Crowley, Dan Gunter, Jason Lee, and Mary Thompson. A monitoring sensor management system for grid environments. *Cluster Computing*, 4(1):19–28, 2001.
242. Hong-Linh Truong. SISPROFILING User's Guide. Technical report, Institute for Software Science, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria, Jan 2002.
243. Hong-Linh Truong and Thomas Fahringer. SCALEA Version 1.0: User's guide. Technical report, Institute for Software Science, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria, April 2001.
244. Hong-Linh Truong and Thomas Fahringer. SCALEA: A Performance Analysis Tool for Distributed and Parallel Program. In *8th International Europar Conference(EuroPar 2002)*, LNCS 2400, Paderborn, Germany, August 2002. Springer-Verlag.
245. Hong-Linh Truong and Thomas Fahringer. On Utilizing Experiment Data Repository for Performance Analysis of Parallel Applications. In *9th International Europar Conference(EuroPar 2003)*, LNCS 2790, pages 27 – 37, Klagenfurt, Austria, August 2003. Springer-Verlag.
246. Hong-Linh Truong and Thomas Fahringer. SCALEA: A Performance Analysis Tool for Parallel Programs. *Concurrency and Computation: Practice and Experience*, 15(11-12):1001–1025, 2003.

247. Hong-Linh Truong and Thomas Fahringer. A Soft Computing-based Approach to Performance Analysis of Parallel and Distributed Programs. November 2004. On submission.
248. Hong-Linh Truong and Thomas Fahringer. Performance Analysis, Data Sharing and Tools Integration in Grids: New Approach based on Ontology. In *Proceedings of International Conference on Computational Science (ICCS 2004)*, LNCS 3038, pages 424 – 431, Krakow, Poland, Jun 7-9 2004. Springer-Verlag.
249. Hong-Linh Truong and Thomas Fahringer. SCALEA-G: a Unified Monitoring and Performance Analysis System for the Grid. *Scientific Programming*, 12(4):225–237, 2004. IOS Press.
250. Hong-Linh Truong and Thomas Fahringer. SCALEA-G: a Unified Monitoring and Performance Analysis System for the Grid. In *Proceedings of 2nd European Across Grids Conference (AxGrids 2004)*, LNCS 3165, pages 202–211, Nicosia, Cyprus, Jan 28-30 2004. Springer-Verlag.
251. Hong-Linh Truong and Thomas Fahringer. Online Performance Monitoring and Analysis of Grid Scientific Workflows. In *Proceedings of European Grid Conference 2005 (EGC2005)*, LNCS, Amsterdam, The Netherlands, February 14 -16, 2005. Springer-Verlag.
252. Hong-Linh Truong and Thomas Fahringer. Self-Managing Sensor-based Middleware for Performance Monitoring and Data Integration in Grids. In *Proceedings of 19 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, Colorado, USA, April 4-8, 2005. IEEE Computer Society Press.
253. Hong-Linh Truong, Thomas Fahringer, and Schahram Dustdar. Dynamic Instrumentation, Performance Monitoring and Analysis of Grid Scientific Workflows. *Journal of Grid Computing (Kluwer)*, April 2005. On submission, revised version.
254. Hong-Linh Truong, Thomas Fahringer, Michael Geissler, and Georg Madsen. Performance Analysis for MPI Applications with SCALEA. In *Proceedings of 9th European PVM/MPI User's Meeting (EuroPVM/MPI 2002)*, LNCS 2474, Linz, Austria, Sep 29th - Oct 2nd 2002. Springer-Verlag.
255. Hong-Linh Truong, Thomas Fahringer, Georg Madsen, Allen D. Malony, Hans Moritsch, and Sameer Shende. On Using SCALEA for Performance Analysis of Distributed and Parallel Programs. In *Proceedings of the 9th IEEE/ACM High-Performance Networking and Computing Conference (SC'2001)*, Denver, USA, November 2001.
256. Hong-Linh Truong, Thomas Fahringer, Francesco Nerieri, and Schahram Dustdar. Performance Metrics and Ontology for Describing Performance Data of Grid Workflows. In *Proceedings of 1st International Workshop on Grid Performability, held in conjunction with IEEE International Symposium on Cluster Computing and Grid 2005*, Cardiff, UK, 9 - 12 May 2005. IEEE Computer Society Press.
257. M. Tubaishat and S. Madria. Sensor networks: an overview. *IEEE Potentials*, 22(2):20–23, April-May 2003.
258. T. Cortes V. Pillet, J. Labarta and S. Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In *WoTUG-18*, pages 17–31, Manchester, April 1995.
259. Aad J. van der Steen and Jack J. Dongarra. http://www.phys.uu.nl/ euroben/reports/web04/overview.html.

260. Veridian System. *DEEP: Development Environment For Parallel Programs*, April 2001. Also avaible at http://www.psrv.com/.
261. Jeffrey Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *Conference Proceedings of the 2000 International Conference on Supercomputing*, pages 245–254, Santa Fe, New Mexico, May 8–11, 2000. ACM SIGARCH.
262. Mladen A. Vouk and Munindar P. Singh. Quality of service and scientific workflows. Technical Report TR-96-19, Department of Computer Science, North Carolina State University, June 20 1996. Thu, 19 Sep 96 22:59:36 GMT.
263. Fredrik Vraalsen, Ruth A. Aydt, Celso L. Mendes, and Daniel A. Reed. Performance contracts: Predicting and monitoring grid application behavior. In *Proceedings of GRID 2001*, volume LNCS 2242, pages 154–165, Denver, Colorado, Nov 2001. Springer-Verlag.
264. W3C: Web Services Architecture. http://www.w3.org/tr/ws-arch/.
265. J. Wainer, M. Weske, G. Vossen, and C. Bauzer Medeiros. Scientific Workflow Systems. In *Proc. NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, Athens, Georgia, May 1996.
266. OpenMP Website. http://www.openmp.org.
267. Von Welch, Frank Siebenlist, Ian Foster, John Bresnahan, Karl Czajkowski, Jarek Gawor, Carl Kesselman, Sam Meder, Laura Pearlman, and Steven Tuecke. Security for Grid Services. In *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, pages 48–57, Seattle, Washington, June 22 - 24 2003.
268. Felix Wolf and Bernd Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP-11)*, pages 13–22. IEEE Computer Society Press, February 2003.
269. R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems*, 15:757–768, 1999.
270. Extensible Markup Language (XML). http://www.w3.org/xml.
271. XML Schema. http://www.w3.org/xml/schema.
272. Jerry C. Yan. Performance tuning with aims - an automated instrumentation and monitoring system for multicomputers. In *HICSS (2)*, pages 625–633, 1994.
273. L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, June 1965.
274. L. A. Zadeh. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-3(1):28–44, January 1973.
275. L. A. Zadeh. Possibility theory and soft data analysis. In L. M. Cobb and R. M. Thrall, editors, *Mathematical Frontiers of the Social and Policy Sciences*, A.A.A.S Selected Symposium, Volume 54, pages 69–129. Westview Press, Boulder, CO, 1981.
276. L. A. Zadeh. *Fuzzy logic technology and applications*, chapter Preface. IEEE technology updates. IEEE Press, New York, 1994.
277. L. A. Zadeh. Fuzzy Logic = Computing with Words. *IEEE Transactions on Fuzzy Systems*, 4(2):103–111, 1996.
278. Lotfi A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Commun. ACM*, 37(3):77–84, 1994.

279. Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, Fall 1999.
280. H. J. Zimmermann. *Fuzzy Set Theory and its Applications*. Kluwer Academic Press, Boston, MA, 2nd edition, 1993.

# Index