ASKALON: A Tool Set for Cluster and Grid Computing[†]

Thomas Fahringer[‡¶], Alexandru Jugravu[§], Sabri Pllana[§],
Radu Prodan[§], Clovis Seragiotto Junior[§], Hong-Linh Truong[§]

[‡] *Institute for Computer Science, University of Innsbruck,*
Technikerstr. 25/7, A-6020 Innsbruck, Austria
E-mail: Thomas.Fahringer@uibk.ac.at
[§] *Institute for Software Science, University of Vienna,*
Liechtensteinstr. 22, A-1090 Vienna, Austria
E-mail: {aj,pllana,radu,clovis,truong}@par.univie.ac.at

SUMMARY

Performance engineering of parallel and distributed applications is a complex task that iterates through various phases, ranging from modeling and prediction, to performance measurement, experiment management, data collection, and bottleneck analysis. There is no evidence so far that all of these phases should/can be integrated in a single monolithic tool. Moreover, the emergence of computational Grids as a common single wide-area platform for high-performance computing raises the idea to provide performance tools and others as interacting Grid services that share resources, support interoperability among different users and tools, and most important provide omni-present performance functionality over the Grid.

We have developed the ASKALON tool set [18] to support performance-oriented development of parallel and distributed (Grid) applications. ASKALON comprises four tools, coherently integrated into a Grid service-based distributed architecture. SCALEA is a performance instrumentation, measurement, and analysis tool of parallel and distributed applications. ZENTURIO is a general purpose experiment management tool with advanced support for multi-experiment performance analysis and parameter studies. AKSUM provides semi-automatic high-level performance bottleneck detection through a special-purpose performance property specification language. The PerformanceProphet enables the user to model and predict the performance of parallel applications at early development stages.

In this paper we describe the overall architecture of the ASKALON tool set and outline the basic functionality of the four constituent tools. The structure of each tool is based on the composition and sharing of remote Grid services, thus enabling tool interoperability. In addition, a common Data Repository allows the tools to share common application performance and output data which has been derived by the individual tools. A Service Repository is used to store common portable Grid service implementations. A general-purpose Factory service is employed to create service instances on arbitrary remote Grid sites. Discovering and dynamically binding to existing remote services is achieved through Registry services. Visualization is supported by the ASKALON visualization diagrams in order to graphically display performance and output data by querying the Data Repository.

We demonstrate the usefulness and effectiveness of ASKALON by applying the tools to a variety of real-world applications.

[†] This research is supported by the Austrian Science Fund as part of the Aurora Project under contract SFBF1104.
[¶] Corresponding author

1. Introduction

Computational Grids have become an important asset aiming at enabling application developers to aggregate resources scattered around the globe for large-scale scientific and engineering research. However, developing applications that can effectively utilize the Grid still remains very difficult due to the lack of high-level tools to support developers. To this date, many individual efforts have been devoted to support performance-oriented development of parallel and distributed applications. Porting existing software tools on the Grid poses additional challanges with respect to portability and interoperability for concurrent use of shared resources. Portability and interoperability of software tools on the Grid are critical issues which have not been thoroughly addressed by the scientific community. We believe that this situation has been caused by the heterogeneous and often machine-dependent nature of tools, complex operating system and compiler dependencies, as well as differences and incompatibilities in tool functionality, interfaces, and other proprietary solutions.

Since 1995, the Globus project [23] is continuously developing middleware technology aimed to support and ease the development of high-level Grid infrastructures and applications. Despite its enormous success in the Grid research community, the Globus Toolkit Version 2 suffers from substantial integration and deployment problems. The services provided are largely independent from each other. The only clear connection among is the common Grid Security Infrastructure (GSI) [25]. Therefore, improvements made by the community to one service caused little or no contributions to others, thus slowing down progress. Moreover, the implementation platform has been mostly based on C, which limits the deployment and software reuse capabilities.

While language, software, system, and network neutrality have not been initially on the Globus agenda, they have been successfully addressed over the past 10 years by well known distributed object-oriented component technologies such as the Java Remote Method Invocation (RMI [29]), the Common Object Request Broker Architecture (CORBA [35]), Microsoft's Distributed Component Object Model (DCOM [8]), Enterprise Java Beans [45], Jini [17], Web services [56], or JavaSymphony [21].

In the year 2000, a consortium of companies comprising Microsoft, IBM, BEA Systems, and Intel defined *Web services*, a new set of XML [30] standards for programming Business-to-Business (B2B) applications. Web services are nowadays being standardised by the W3C consortium [56]. They address heterogeneous distributed computing by defining techniques for describing software components, methods for accessing them, and discovery methods that enable the identification of relevant service providers. A key advantage of Web services over previous distributed technology approaches is their programming language, model, network, and system software neutrality.

Follwing the advantages offered by Web services, the Open Grid Services Architecture (OGSA) [24] builds on the Web services technology mechanisms to uniformly expose Globus Grid services semantics, to create, name, and discover transient Grid service instances, to provide location transparency and multiple protocol bindings for service instances, and to support integration with underlying native platform facilities. The Open Grid Services Infrastructure (OGSI) [54] is the technical specification which defines extensions and specialisations to the Web services technology to standardise and ease the development of Grid services as required by OGSA. The OGSA toolkit implements the OGSI specification as an extension to Apache Axis [27].

In this paper we describe the ASKALON tool set for cluster and Grid computing [18]. ASKALON integrates four interoperable tools: SCALEA for instrumentation and performance analysis, ZENTURIO for automatic experiment management, AKSUM for automatic bottleneck analysis, and the PerformanceProphet for application modeling and performance prediction. The tool-set has been designed as a distributed set of (OGSI-based) Grid services, exporting a platform independent standard API. Platform dependent and proprietary services are pre-installed on specific appropriate sites and can be remotely accessed through a portable interface. A UDDI-based service repository is employed to store implementations of public portable Grid services. Each tool provides its own graphical User Portal to be accessed by the user in a friendly and intuitive way. Remote services are created by a general purpose *Factory* service using the information from the Service Repository. On the other hand, the portals discover and bind to existing service instances by means of advanced lookup operations invoked on the *Registry* service. Interoperability between tools is naturally achieved by allowing multiple clients to connect and share the same service instances from the initial design phase. Furthermore, a Data Repository with a standard schema definition, allows tools to share performance and output data of the applications under evaluation.

The paper is organized as follows. The next section discusses related work. Section 3 presents an overall Grid service-based architecture of the ASKALON tool-set. Sections 4 through 7 describe the basic functionality of each tool in brief. Various experiments conducted by each individual tool on several real world applications are reported in Section 8. Concluding remarks and a brief future work outline are presented in Section 9.

2. Related Work

Early work at the Technical University of Munich developed THE TOOL-SET [57], consisting of a mixture of performance analysis and debugging tools for parallel computing. Attempts to accomodate these tools into a single coherent environment produced the On-line Moniotoring Interface Specification (OMIS) [58]. In contrast to this effort, ASKALON focuses on performance analysis for parallel and Grid applications, whose tools are integrated through a distributed Grid service-based design.

Significant work on performance measurement and analysis has been done by Paradyn [38], TAU [37], Pablo toolkit [44], and EXPERT [59]. SCALEA differs from these approaches by providing a more flexible mechanism to control instrumentation for code regions and performance metrics of interest. Although Paradyn enables dynamic insertion of probes into a running code, Paradyn is currently limited to instrumentation of subroutines and functions, whereas SCALEA can instrument – at compile-time only – arbitrary code regions including single statements. Paradyn also supports experiment management [33] through a representation of the execution space of performance experiments and techniques for quantitative comparison of several experiments. In contrast to ZENTURIO, experiments (by varying problem and machine size parameters) have to be set up manually.

The National Institute of Standards and Technology (NIST) developed a prototype for an automated benchmarking tool-set [12] to reduce the manual effort in running and analysing the results of parallel benchmarks. Unlike in ZENTURIO, experiment specification is restricted to pre-defined parameters available through a special purpose graphical user interface.

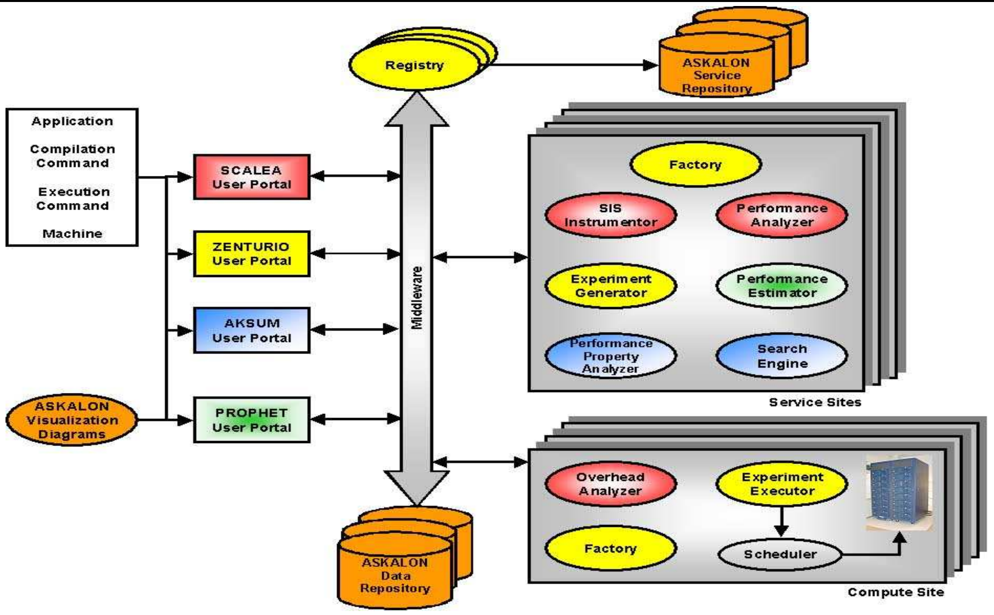


Figure 1. The ASKALON Tool Set Architecture.

The ZOO project [32] has been initiated to support scientific experiment management based on a desktop experiment management environment. Experiments are designed by using an object-oriented data description language. Nimrod [1] and ILAB [60] are two Grid-based tools is a tool that manages the execution of large scale parameter studies. Parameterization is specified by meand of a declarative plan file in Nimrod and through a graphical interface in ILAB. However, ZOO, Nimrod, and ILAB restrict parameter specification to input files only, in contrast to ZENTURIO, which allows parameter specification within arbitrary application files.

The European working group APART [4] defined a specification language for performance properties of parallel programs based on which JavaPSL, the language for performance property specification used in AKSUM, has been designed. Performance properties defined by APART also inspired some of the predefined properties Aksum provides.

The POEMS [2] project introduced a graphical representation which captures the parallel structure, communication, synchronization, and sequential behavior of the application. No standards are used to model parallel applications and we have not found any document that describes a graphical representation of process topologies or computer architectures.

3. ASKALON Architecture

ASKALON has been designed as a set of distributed Grid services (see Figure 1). The services are based on the OGSI-technology and expose a platform independent standard API, expressed in the standard Web Services Description Language (WSDL) [11]. Platform dependent and proprietary services are pre-installed on specific appropriate sites from where they can be remotely accessed in a portable way, via the Simple Object Access Protocol (SOAP) [46] over HTTP. By isolating platform dependencies on critical resources, extra flexibility in installing and managing the tools is achieved. Each tool provides its own graphical User Portal to be accessed in a friendly and intuitive way. The User Portals are light-weight clients, easy to be installed and managed by the end-users. User Portals reside on the user's local machine (e.g. a notebook) and provide gateways to performance tools by dynamically creating and connecting to remote services. ASKALON services can be persistent (e.g. Registry and Factory) or transient, as specified by OGSI. All services can be accessed concurrently by multiple clients, which is an essential feature in a Grid environment and enables tool interoperability. The Grid Security Infrastructure (GSI) [26] based on single sign-on, credential delegation, and Web services security [5] through XML digital signature and XML encryption is employed for authentication across ASKALON User Portals and Grid services.

Remote service instances are created by a general-purpose *Factory* service (see Section 3.2 using the information from the Service Repository (see Section 3.1). At the same time, the portals discover and bind to existing service instances using the *Registry* service, described in Section 3.3. Additionally, the *Data Repository* (see Section 3.4) with a common standard schema definition, stores and shares common performance and output data of the applications under evaluation. It thus provides an additional mode of integration and interoperability among tools. To increase reliability of the system by avoiding single point of failures, multiple Registry, Service, and Data Repository instances are replicated on multiple sites and run independently.

An OGSI-based asynchronous event framework enables Grid services to notify clients about interesting system and application events. ASKALON services support both push and pull event models, as specified by the Grid Monitoring Architecture (GMA) [50]. Push events are important for capturing dynamic information about running applications and the overall Grid system on-the-fly and avoids expensive continuous polling. Pull events are crucial for logging important information, for instance in cases when tools like ZENTURIO run in off-line mode, with disconnected off-line users.

ASKALON classifies the Grid sites on which the services can run into two categories (see Figure 1):

(1) *Compute sites* are Grid locations where end applications run and which host services intimately related to the application execution. Such services include the Experiment Executor of ZENTURIO, in charge of submitting and controlling jobs on the local sites and the Overhead Analyzer of SCALEA, which transforms raw performance data collected from the running applications into higher-level more meaningful performance overheads.

(2) *Service sites* are arbitrary Grid locations on which ASKALON services are pre-installed or dynamically created by using the Factory service.

3.1. UDDI-based Service Repository

The Universal Description, Discovery, and Integration (UDDI) [55] is a specification of a centralized registry for publishing persistent Web service instances. UDDI as a static repository (a database) is obviously not appropriate for publishing transient Grid service instances according to the UDDI best practices document [13]. At the same time, OGSI defines the Factory `portType`, but omits to define how the Factory accesses the implementation code of the service to be created.

Publishing service implementations in a Grid environment is crucial, as one cannot assume that a service implementation is available at the site where it is supposed to execute. This assumption becomes reasonable for deployment scenarios of portable Java code, which is the case of the ASKALON service implementations. We therefore propose a slightly different usage of UDDI in a Grid environment, which, instead of publishing dynamic transient Grid service instances, uses UDDI to publish static persistent Grid service implementations.

Our Factory service downloads the required service implementation from the UDDI Registry (if necessary) and deploys the service instance within its local hosting environment.

Technically speaking, the UDDI best practices requires that the interface part of the WSDL document be published as a UDDI `tModel` and the instance part as a `businessService` element (as URLs). The `businessService` UDDI element is a descriptive container, which is used to group related Web services. It contains one or more `bindingTemplate` elements, which contain information for connecting and invoking a Web service. The `bindingTemplate` contains a pointer to a `tModel` and an `accessPoint` element, which is set to the SOAP address of the service (`port`). In contrast, for Web services running in a Grid environment, we use the `businessService` element to publish service implementation information. Therefore, we set the `accessPoint` element of a `bindingTemplate` with the URL to the JAR package that implements the service.

Registry and Factory are the only persistent services in ASKALON, for which one entry corresponding to the service implementation and an arbitrary number of entries corresponding to the available instances are published in the UDDI repository. The distinction between the two types of entries is done through the syntax of the `accessPoint` URL. The User Portal connects to all available Registries at start-up time, while Factories have a standard URL derived from the host name and a pre-defined port number (`http://hostname:port/Factory/`). Additionally, a notification mechanism compliant with the newest UDDI Version 3 specification informs the portal when new Registries are registered.

3.2. Factory

Each Grid site hosts by default one persistent Factory service. The Factory is a generic service that creates and deploys (Java) Grid services of any type at run-time, implemented and packaged as JAR files. The Factory searches the (UDDI) Service Repository for a service of a given type (as `businessService` name). If such a service is found, the Factory creates a Grid service instance on-the-fly through the following steps: (1) get the URL of the service implementation (represented as `accessPoint` element – see Section 3.1); (2) download the JAR package; (3) deploy the service in the hosting environment in which the Factory resides; (4) initialize the service instance; (5) register the instance with all Registries; (6) set a leasing time equal to the service termination time; (7) return the URL to the WSDL file of the service instance. Clients use this URL to retrieve the WSDL file and dynamically bind to the service through dynamic run-time generated proxies.

3.3. Registry

The Registry service is a persistent service which maintains an updated list of the URLs to the WSDL files of the existing Grid service instances. It may reside anywhere in the system and there can be an arbitrary number of Registries for registering Grid services. The User Portal discovers all existing Registries from the Service Repository at start-up time. The Registry grants *leases* to registered services, similar to the Jini [17] built-in leasing mechanism. If a service does not renew its lease before the lease expires, the Registry deletes it from its service list. This is an efficient way to cope with dynamic transient services and network failures. A leasing time of 0 seconds explicitly unregisters the service. An event mechanism informs clients about new services that registered with the Registry, or when services did not renew their lease. Thereby, clients are always provided with a dynamically updated view of the environment.

We provide three methods for performing Registry lookup operations: (1) *white pages* provide service discovery based on service URL; (2) *yellow pages* support service discovery based on service type. (3) *green pages* perform discovery based on service functionality expressed in WSDL (see [43] for details).

3.4. Data Repository

All ASKALON tools share a common data repository for storing information about the parallel and distributed applications under evaluation. The repository implementation is based on the PostgreSQL [31] open-source relational database system. The database schema definition reflects a layered design and has been jointly developed by all tool developers.

Any tool can optionally store relevant experimental data including application, source code, machine information, and performance and output results into the repository. An interface with search and filter capabilities for accessing repository and leveraging the performance data sharing and tool integration [52] is provided. Tools exchange data via the data repository and also provide direct interfaces to subscribe for specific performance metrics, or parameter study results. Data can also be exported into XML format so that it can easily be transfered to and processed by other tools.

SCALEA stores mostly performance overheads, profiles and metrics in the the data repository. ZENTURIO through the Experiment Executor adds information about experiment parameters (ZEN variables) as well as output data required by parameter studies. AKSUM adds through its Property Analyzer the ZENTURIO schema definition information about high-level performance properties (inefficiency, scalability) and their severity. The PerformanceProphet can access information provided by any ASKALON tool to guide its prediction effort. Moreover, predicted performance data can be inserted into the data repository as well, which can be accessed by ZENTURIO and AKSUM instead of invoking SCALEA for a real program run.

3.5. ASKALON Visualization Diagrams

In addition to the distributed Grid service-based design and the common Data Repository, ASKALON provides a Java-based package consisting of a set of generic and customizable set of visualization diagrams [20]. Available diagrams include linechart, barchart, piechart, surface, as well as a more sophisticated hierarchical diagram for the simultaneous visualization of a maximum of seven dimensions, which is used to graphically display Grid performance studies.

Besides visualizing static post-mortem information, all diagrams accept online data streams as input for dynamic on-line visualization of parallel and distributed program behavior. The diagrams are generic and fully customizable, which enables both user and Grid services to map application parameters, output results, or performance metrics onto arbitrary visualization axis. All ASKALON tools employ this diagram package for visualization.

4. SCALEA

SCALEA [53] is a performance instrumentation, measurement, and analysis tool for parallel programs that supports post-mortem performance analysis.

4.1. Instrumentation

The Instrumentation service provides support to instrument services and applications. We support two approaches: source code and dynamic instrumentation. In the first approach, the SCALEA Instrumentation System (SIS) supports automatic instrumentation of Fortran MPI, OpenMP, HPF, and mixed OpenMP/MPI programs. The user is able to select (by directives or command-line options) code regions and performance metrics of interest. Moreover, SIS offers 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 analyzes source codes and inserts probes (instrumentation code) in the code which will collect all relevant performance information during execution of the program on a target architecture.

The source code level approach, however, requires all the source files to be available. In addition, instrumentation and measurement metrics can not be configured at runtime. To overcome these problems, we are currently exploiting the dynamic instrumentation mechanism based on Dyninst [10]. In order to enable dynamic instrumentation, we implement a *mutator service* which contains Dyninst API calls, the code that implements the runtime compiler and the utility routines to manipulate the application process. A mutator is responsible for controlling the instrumentation of an application process on the machine where the process is running. We developed an XML-based instrumentation request language (IRL) to allow users and services to specify code regions for which performance metrics should be determined and to control the instrumentation process.

4.2. Overhead Analyzer

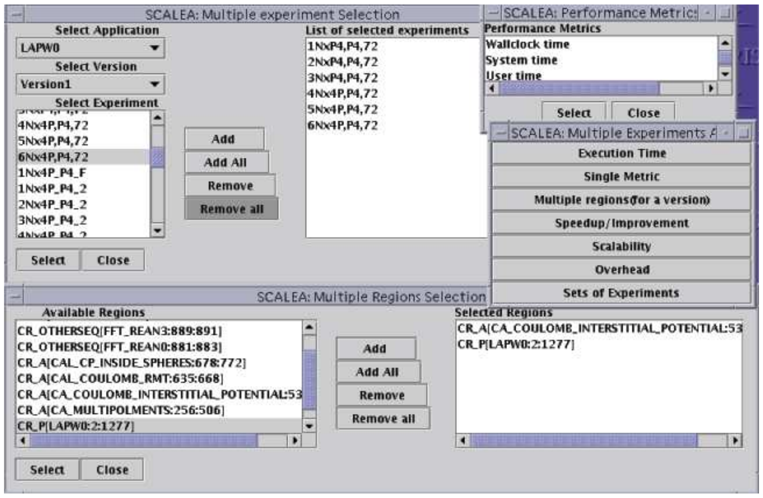
SCALEA provides a novel classification of performance overheads for parallel programs that includes data movement, synchronization, control of parallelism, additional computation, loss of parallelism, and unidentified overheads [53]. The Overhead Analyzer service is used to investigate performance overheads of a parallel or distributed program based on the overhead classification.

The SIS measurement library supports profiling of parallel applications, collecting timing, counter information, as well as hardware parameters via the PAPI library [9]. The Overhead Analyzer computes performance overheads and stores them into the data repository.

4.3. Performance Analyzer

The Performance Analyzer service evaluates the raw performance data collected during program execution and stores them into the data repository. All requested performance metrics are computed. Several analyses (e.g. *Load Imbalance Analysis, Inclusive/Exclusive Analysis, Metric Ratio Analysis, Overhead Analysis, Summary Analysis*) are provided.

While most performance tools investigate the performance for individual experiments one at a time, SCALEA goes beyond this limitation by supporting also performance analysis for multiple



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 Metric

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]

Figure 2. SCALEA Multiple Experiments Analysis

experiments (e.g. *Speedup/Improvement Analysis*, *Scalability Analysis*, *Multi-Region Analysis*, *Multi-Set Experiment Analysis*). The user can select several experiments, code regions and performance metrics of interest whose associated data are stored in the data repository (see Figure 2). The outcome of every selected metric is then analyzed and visualized for all experiments. SCALEA supports the following multi-experiment analyses:

- **performance comparison for different sets of experiments**: The overall execution of the application across different sets of experiments can be analyzed; experiments in a set are grouped based on their characteristics (e.g. problem sizes, communication libraries, platforms). .
- **overhead analysis for multi-experiments**: Various sources of performance overheads across experiments can be examined.
- **parallel speedup and efficiency at both program and code region level**: Commonly, those metrics are applied only at the level of the entire program. SCALEA, however, supports examination of scalability at both program and code region level ranging from a single statement to the entire program.

5. ZENTURIO

ZENTURIO [43] is a tool designed to specify and automatically conduct large sets of experiments in the context of large scale performance and parameter studies on cluster and Grid architectures.

5.1. ZEN Experiment Specification Language

Conventional parameter study tools [1, 60] restrict parameterization to input files only. In contrast, ZENTURIO defines a directive-based language called ZEN [42] to annotate arbitrary application files. ZEN directives are used to assign value sets to so called *ZEN variables*. A ZEN variable can represent any problem, system, or machine parameter, including program variables, file names, compiler options, target machines, machine sizes, scheduling strategies, data distributions, etc. The value set represents the list of interesting values for the corresponding parameter. The advantage of the directive-based approach over an external script [1] is the ability to specify more detailed experiments (e.g. associate local scopes to directives, restrict parametrization to specific local variables, evaluate different scheduling alternatives for individual loops, etc.).

ZEN defines four kinds of ZEN directives, exemplified in Section 8.3 The *ZEN substitute directive* (see Examples 8.1, 8.2) uses a pre-processor-based string replacement mechanism to substitute a ZEN variable with one element from its value set. To complement some of the ZEN substitute directive's limitations, a *ZEN assignment directive* (see Example 8.3) can be used to insert an assignment statement in the code. By default, the cross product of the value sets of all ZEN variables is computed. The number of possible value set combinations can be limited by means of *ZEN constraint directives* (see Example 8.3). ZEN also supports the specification of performance metrics for arbitrary code regions through the *ZEN performance behavior directives* (see Example 8.2). A file/application annotated with ZEN directives is called ZEN file/application. A ZEN transformation system generates all ZEN file instances for a ZEN file, based on the ZEN directives inserted. The SCALEA instrumentation engine, which is based on a complete Fortran90 OpenMP, MPI, and HPF front-end and unparser, is used to instrument the application for performance metrics. The ZEN performance behavior directives are translated to SCALEA SIS directives and compiler command-line options.

5.2. Experiment Generator

Through the Registry service, the User Portal locates an *Experiment Generator* service to which it transfers the entire application via GridFTP. Designing Experiment Generator as a separate service has the advantage of isolating the platform dependencies of the Vienna Fortran Compiler (VFC) [6], on which the SCALEA instrumentation engine is based. After experiments are generated, they are automatically transferred to the target compute Grid sites (using GridFTP [3]) for execution. In case of the Globus DUROC co-allocator [15], the experiments are copied to multiple destination sites, which are read from the RSL (Resource Specification Language) description of the experiment.

5.3. Experiment Executor

The *Experiment Executor* is a generic service responsible for compiling, executing, and managing the execution of experiments onto the local Grid site. If no Experiment Executor exists on the execution site (arbitrary site in case of DUROC), it is created by the Factory service. The Experiment Executor interacts at the back-end with a batch job scheduler, which in the current implementation can be Condor [36], LoadLeveler, LSF, PBS, and Sun Grid Engine for cluster, and GRAM [14] or DUROC for Grid computing.

After each experiment has completed, the application output results and performance data are stored into the ASKALON Data Repository (see Section 3.4). High-level performance overheads are computed by the Overhead Analyzer service of SCALEA. An *Application Data Visualizer* portlet of the User Portal, developed on top of the ASKALON visualization package (see Section 3.5), automatically generates visualization diagrams that display the variation of performance and output data across multiple sets of experiments.

6. AKSUM

AKSUM is a flexible tool for semiautomatic multi-experiment performance analysis of parallel and distributed applications [19]. Through its User Portal, the user inputs hypotheses that should be tested, machine and problem sizes for which performance analysis should be done (application input parameters), files that compound the application, and possibly conditions to stop the analysis process. The User Portal displays, while the search process is going on, which hypotheses were evaluated to true for the machine and problem sizes tested.

In AKSUM, hypotheses are called performance properties; each performance property is an algorithm specifying a negative performance behavior that may be found in a program. The specification may be very simple (for example, reflecting directly some measurement made by SCALEA), but it can also characterize a very complex property, like replicated code or the inefficiency of a set of experiments. Users can use the set of properties AKSUM provides and also write, in Java, their own properties and add them to AKSUM in order to extend it. Any new property must be a class defining the following three methods:

- *boolean holds( )*: returns true if the property (class) instance holds (that means, the ”negative performance behavior” is present).
- *float getSeverity( )*: returns a value between 0 and 1 indicating how severe a property instance is (the closer to 1, the more severe the property instance is).
- *float getConfidence( )*: returns a value between 0 and 1 that indicates the degree of confidence in the correctness of the value returned by *holds.*

Aksum comes with a library, called JavaPSL [22], to help with the specification of performance properties, as it allows easy access to the performance data (timing information, overheads and hardware counters) that SCALEA provides.

6.1. Services: Search Engine, Instrumentation Engine and Performance Property Analyzer

The user-supplied input data is provided to the *Search Engine*, which is in the center of AKSUM and controls the entire search process. By issuing requests to the *Instrumentation Engine*, the Search Engine determines the performance information to be collected for application code regions and problem and machine sizes. The Instrumentation Engine of AKSUM invokes the SCALEA Instrumentation service for the actual code instrumentation, that is, it is a layer that enables the Search Engine to access and traverse application files in a machine independent way, to instrument them, and transparently to modify makefiles, scripts, and the compilation command line in order to link the instrumented application with the instrumentation library provided by SCALEA.

The instrumented code is submitted to ZENTURIO's Experiment Generator service, which change the execution parameters according to the input parameters provided by the user and transfer the files to the appropriate Grid sites where the ZENTURIO's Experiment Executor service will compile and execute the experiments, as well as transfer performance data to the Data Repository after each experiment has been executed.

The Search Engine evaluates the performance data in the Data Repository by invoking the *Performance Property Analyzer* service, which determines all critical performance properties (that is, property instance whose value returned by the method *getSeverity* is greater than a certain threshold). A cycle consisting of consecutive phases of application execution and property evaluation is continued until all experiments are done or some system or user-defined condition stops the search process. Under the User Portal, every performance property that has been determined to be critical is dynamically displayed (together with the source code) to the user during the search process and stored in the ASKALON Data Repository.

The way performance property instances are ordered and grouped, and which information should be displayed, may be also configured by describing the ”fields” that should compose each record (property instance) in the output. Possible fields for an output record are the code region where a property instance has been found, the experiment associated to the property instance, the number of nodes, processes or threads used in the experiment, the property name, and any of the application input parameters.

7. PerformanceProphet

The PerformanceProphet is a set of performance modeling and prediction services, which supports prediction of the performance behavior of distributed and parallel applications on cluster architectures. At present, the PerformanceProphet does not support prediction for the Grid.

The central idea is to support the application developer by providing performance analysis results at early stages of the application development. For instance, the user can evaluate various parallelization strategies at modeling time before the application is developed, by building a high level model of the application and evaluating its performance. PerformanceProphet consists of two main components: (i) the user portal *Teuta*, which supports the application model development based on the Unified Modeling Language (UML) [39], and (ii) the *Performance Estimator* service, which supports the performance evaluation of an application model, based on a hybrid approach which combines analytical modeling with discrete event simulation.

Figure 3 shows a scenario of usage of PerformanceProphet for performance modeling and prediction of distributed and parallel applications:

1. The user constructs the model of the application by combining UML building blocks (such as OpenMP ParallelDo, see Figure 3) annotated with performance and control flow information by using *Teuta*. The application model is transformed to an intermediate form that is used as input for the *Performance Estimator*, and is stored into the ASKALON Data Repository (see Section 3.4).
2. The *Performance Estimator* reads the model from the ASKALON Data Repository and evaluates the performance of the application for the user-selected target computer architecture. The obtained performance prediction data is stored into the ASKALON Data Repository. Other tools

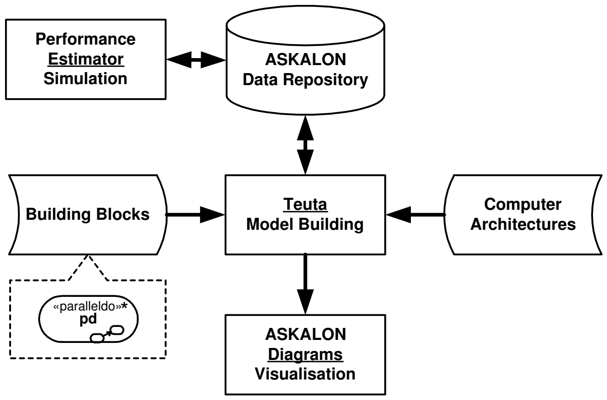


Figure 3. Performance modeling and prediction with PerformanceProphet

(for instance AKSUM and ZENTURIO) can obtain the performance prediction data either via ASKALON Data Repository or directly from *Performance Estimator* service by subscribing for specific performance metrics and overheads.

3. If the user is not satisfied with the result, then he may modify the model of the application and/or select another target computer architecture. This cyclic process of model adaptation and performance prediction may continue until acceptable performance results are obtained.

7.1. UML Extension for the Domain of Performance-Oriented Computing

We have determined that by using only the core UML, some important shared memory and message passing concepts can not be modeled at all or results in complex diagrams [40]. In order to overcome this issue we have developed an extension of UML for the domain of performance-oriented computing. In [41] we describe a set of UML building blocks that model some of the most important constructs of message passing and shared memory parallel paradigms, which can be used to develop models for large and complex parallel and distributed applications. These building blocks have been largely motivated by the OpenMP and MPI standards. In order to provide tool support for the UML extension for the domain of performance-oriented computing we have developed Teuta. The requirements that guided the design of Teuta are extensibility and platform-independence.

It is difficult to foresee all the types of the building blocks that the user might want to use to model his application. Therefore, we have included a basic set of building blocks, which are described in [41], and made it easy to extend the tool with the new building blocks. The extensibility requirement led us to the use of XML for several tasks, such as the definition of the modeling elements and the configuration of the tool.

Because the application developers may work on various platforms, the tool should be able to run on various platforms as well. The requirement for platform independence led us to the use of the Java programming language for the implementation of Teuta based on the Model-View-Controller (MVC) paradigm [34]. MVC is a paradigm that enforces the separation between the user interface and the rest of the application.

7.2. Hybrid Model Building and Simulation

Hybrid simulation models, which combine analytical modeling techniques with discrete event simulation, present an efficient alternative to simulation-only models [48]. Here we briefly describe our approach of hybrid application model building. A distributed and parallel application consists of two classes of building blocks:

1. Building blocks which involve one *processing unit* (a process or thread), for instance the execution of a sequence of computational operations.
2. Building blocks which involve multiple processing units, for instance collective communication operations, barriers, semaphores, and so on. This type of building blocks impose synchronization among the processing units.

We use analytic modeling techniques to model the execution time of the first type of building blocks by parameterized cost functions. On the other hand, for the modeling of performance behavior of the second type of building blocks we use simulation.

Usually, the cost functions for building blocks are provided by the application developer. Alternatively, the ASKALON tool set may be utilized to develop the cost functions. The first step is to instrument the parts of the application code that correspond to building blocks of interest by using SCALEA Instrumentation Service (see Section 4). Then the set of tests to be executed is defined by using ZEN Experiment Specification Language (see Section 5). The set of tests is defined by the application input parameters and the parameters of computer architecture by using Experiment Generator service of ZENTURIO. Then PerformanceProphet utilizes the Experiment Executor service of ZENTURIO to perform all the defined experiments, and store the performance data into the ASKALON Data Repository. PerformanceProphet makes use of the collected performance data to generate the cost functions based on regression statistical method.

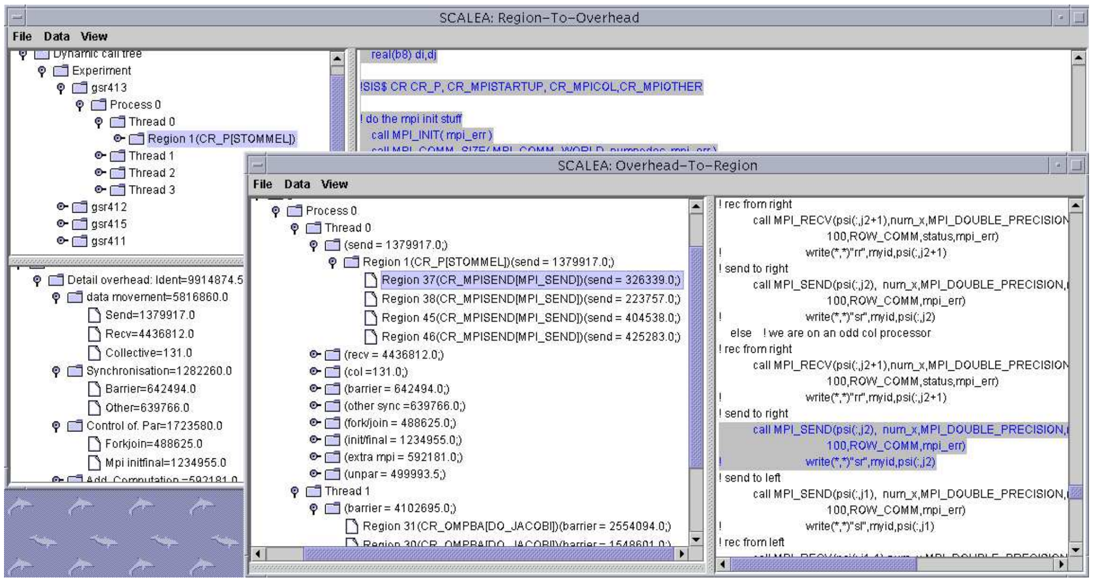
8. Experiments

In this section, we present numerous experiments to demonstrate the usefulness and effectiveness of the ASKALON tool set for a variety of real world applications.

8.1. Overhead Analysis with SCALEA

We illustrate SCALEA by applying it to a mixed OpenMP/MPI Fortran program that solves the 2d Stommel model [49] of an ocean circulation using a five-point stencil and Jacobi iteration.

By using SCALEA we examine the performance overheads for a single experiment of a given program by providing two modes for this analysis. Firstly, the *Region-to-Overhead* mode (see the “Region-to-Overhead” window in Fig. 4) allows us to select any code region instance in the dynamic code region call graph (DRG) [53] for which all detected performance overheads are displayed. Secondly, the *Overhead-to-Region* mode (see the “Overhead-to-Region” window in Fig. 4) enables us to select the performance overhead of interest, based on which SCALEA displays the corresponding code region(s) in which this overhead occurs. This selection can be limited to a specific code region



```
Dynamic call tree
  Experiment
    gsr413
      Process 0
        Thread 0
          Region 1(CR_P[STOMMEL])
        Thread 1
        Thread 2
        Thread 3
    gsr412
    gsr415
    gsr411
```

```
  real(b8) di,dj

!SIS$ CR CR_P, CR_MPISTARTUP, CR_MPICOL,CR_MPIOTHER

! do the mpi init stuff
  call MPI_INIT( mpi_err )
```

```
Detail overhead: Ident=9914874.5
  data movement=5816860.0
    Send=1379917.0
    Recv=4436812.0
    Collective=131.0
  Synchronisation=1282260.0
    Barrier=642494.0
    Other=639766.0
  Control of. Par=1723580.0
    Forkjoin=488625.0
    Mpi initfinal=1234955.0
```

```
Process 0
  Thread 0
    (send = 1379917.0;)
      Region 1(CR_P[STOMMEL])(send = 1379917.0;)
        Region 37(CR_MPISEND[MPI_SEND])(send = 326339.0;)
        Region 38(CR_MPISEND[MPI_SEND])(send = 223757.0;)
        Region 45(CR_MPISEND[MPI_SEND])(send = 404538.0;)
        Region 46(CR_MPISEND[MPI_SEND])(send = 425283.0;)
    (recv = 4436812.0;)
    (col =131.0;)
    (barrier = 642494.0;)
    (other sync =639766.0;)
    (fork/join = 488625.0;)
    (init/final = 1234955.0;)
    (extra mpi = 592181.0;)
    (unpar = 499993.5;)
  Thread 1
    (barrier = 4102695.0;)
      Region 31(CR_OMPBA[DO_JACOBI])(barrier = 2554094.0;)
```

```
! rec from right
        call MPI_RECV(psi(:,j2+1),num_x,MPI_DOUBLE_PRECISION
                      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,
                      100,ROW_COMM,mpi_err)
!                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
                      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,
                      100,ROW_COMM,mpi_err)
!                write(*,*)"sr",myid,psi(:,j2)
! send to left
        call MPI_SEND(psi(:,j1),  num_x,MPI_DOUBLE_PRECISION,
                      100,ROW_COMM,mpi_err)
!                write(*,*)"sl",myid,psi(:,j1)
! rec from left
```

Figure 4. Region-To-Overhead and Overhead-To-Region DRG View.

instance, thread or process. For both modes the source code of a region is shown if the code region instance is selected in the DRG by a mouse click.

8.2. Multiple-Experiment Analysis with SCALEA

In this section we demonstrate a multi-experiment analysis with SCALEA applied to LAPW0 [7], 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.

We use *Multi-Set Experiment Analysis* to study the performance of LAPW0 for two problem sizes and six machine sizes with two different network configurations as shown in Fig. 5. Based on this study, we observed that changing the communication network from Fast-Ethernet by Myrinet did not actually improve the performance.

SCALEA provides a *Performance Overhead Summary* to examine various sources of performance overheads across experiments. For example, the overhead summary for LAPW0 with problem size of 36 atoms displayed in Fig. 6 uncovers a small amount of data movement overhead but a large of overhead for loss of parallelism and unidentified overhead. As a result, instead of focusing our effort on analyzing code regions that are sources of data movement (e.g. send/receive), we study code regions that possibly cause loss of parallelism overhead.

In order to support studying the performance behavior of selected code regions, SCALEA provides a *Multiple Region Analysis*. For instance, the left-window of Fig. 7 visualizes the execution times for the most computational intensive code regions in LAPW0. The right-window of Fig. 7 displays the program's speedup/improvement behavior. The execution times of code regions including CAL_CP_INSIDE_SPHERES, CAL_COULOMB_RMT,

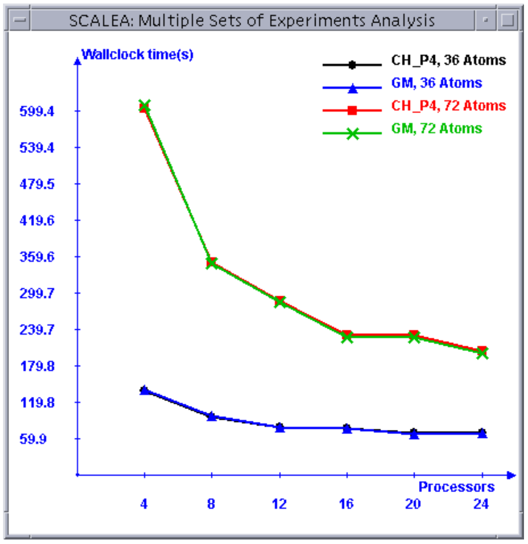
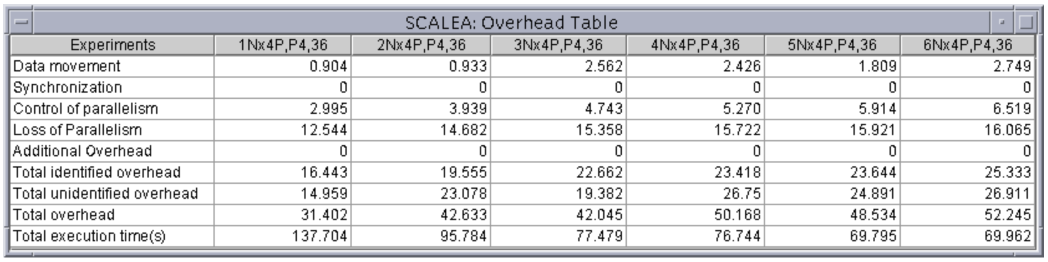


Figure 5. Execution time of LAPW0 with 36 and 72 atoms. *CH_P4, GM* means that MPICH has been used for CH_P4 (for Fast-Ethernet 100Mbps) and Myrinet, respectively.


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

Figure 6. Performance overheads for LAPW0.

CA_COULOMB_INTERSTITIAL_POTENTIAL, CA_MULTIPOLMENTS remain almost constant although the number of processors is increased from 12 to 16 and 20 to 24. In addition, code regions FFT_REAN0, FFT_REAN3, and FFT_REAN4 are executed sequentially. These code regions should therefore be subject of parallelization in order to gain performance.

8.3. Performance and Parameter Studies of Backward Pricing Application with ZENTURIO

The backward pricing kernel [16] is a parallel implementation of the backward induction algorithm which computes the price of an interest rate dependent financial product, such as a variable coupon bond. It is based on the Hull and White trinomial interest rate tree which models future developments of interest rates. We have performed a performance and a parameter study for this code using ZENTURIO.

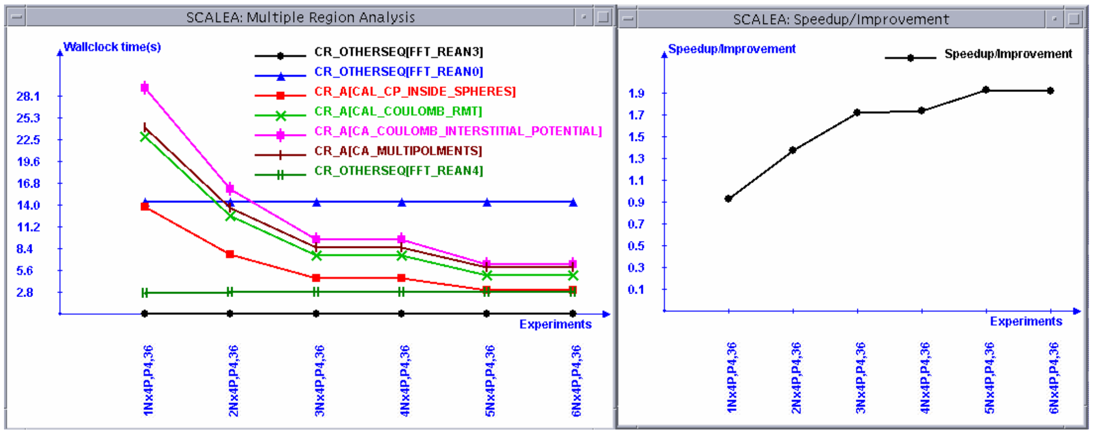


Figure 7. Execution time of computationally intensive code regions (left window) and program's speedup/improvement (right window). *1Nx4P,P4, 36* means 1 SMP node with 4 processors using MPICH CH_P4 and the problem size is 36 atoms.

8.3.1. Performance Study

Backward pricing has been encoded as an HPF+ application which uses HPF+ directives to distribute the data onto the SMP nodes. The application is compiled into a mixed OpenMP/MPI program by the SCALEA instrumentation engine built on top of the HPF+ Vienna Fortran Compiler (VFC). Intra-node parallelization is achieved through OpenMP directives. Communication among SMP nodes is realized through MPI calls. We scheduled the experiments on the SMP cluster using GRAM. We annotated the RSL script to vary the machine size from 1 to 10 SMP nodes (see Example 8.1). The ZEN variable `count=4` is set with the number of SMP nodes. Based on the `count` RSL parameter, GRAM allocates the corresponding number of SMP nodes and uses an available local MPI implementation, which must be defined by the user default shell environment. In the current experiment, we have set our environment for MPICH using the `p4` device over Fast Ethernet. The `MPI_MAX_CLUSTER_SIZE` environment variable ensures that the `mpirun` script starts only one MPI process per SMP node. The intra-node parallelization is achieved by means of OpenMP. We vary from 1 to 4 the number of threads to be forked by an OpenMP `PARALLEL` loop through a ZEN substitute directive. (see Example 8.2). The overall execution time, together with the (MPI) communication and the control of parallelism (HPF+ inspector/executor) overheads have been measured through a ZEN performance behavior directive (see Example 8.2).

Example 8.1 (Globus RSL script)
```
+(&
 (resourceManagerContact="gescher/jobmanager-pbs")
(*ZEN$ SUBSTITUTE count\=4 = {count={1:10}}*)
 (count=4)
 (jobtype=mpi)
 (environment=(MPI_MAX_CLUSTER_SIZE 1))
 (directory="/home/radu/APPS/HANS")
 (executable="bw_halo_sis") )
```

Example 8.2 (Source file)

```
!ZEN$ CR CR_A PMETRIC WTIME, ODATA, OCTRP
. . .
!ZEN$ SUBSTITUTE NUM_THREADS\(4\) = { NUM_THREADS({1:4}) }
!$OMP PARALLEL NUM_THREADS(4)
...
!$OMP END PARALLEL
```

Two ZEN directives have been inserted into two files to produce 40 experiments automatically conducted by ZENTURIO. Figure 8(a) shows a good scalability of this code. Backward pricing is a computational intensive application, which highly benefits from the inter-node MPI and intra-node OpenMP parallelization. The overall wallclock time of the application significantly improves by increasing the number of nodes and OpenMP threads per SMP node. Figure 8(b) shows a very high ratio between the application total execution user time (one full bar) and the HPF and MPI overheads measured, which explains the good parallel behavior. This ratio decreases for a high number of SMP nodes, for which the overheads significantly degrade the overall performance.

8.3.2. *Parameter Study*

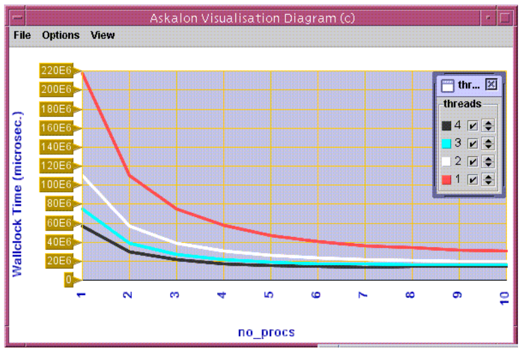
We performed a large parameter study for the Backward pricing code by varying four input parameters: (1) the coupon bond (ZEN variable `coupon` from 0.01 to 0.1 with increment 0.001); (2) the number of time steps, over which the price is computed (ZEN variable `nr_steps` from 5 to 60 with increment 5); (3) the coupon bond's end time (ZEN variable `bond%end`), which must be equal to the number of time steps; (4) the length of one time step (ZEN variable `delta_t` from 1/12 to 1 with increment 1/12). The application has been encoded such that it reads its input parameters from different input data files. ZEN assignment directives are inserted in the source code immediately after the corresponding `read` statements (see Example 8.3). Thus, the original `read` statement is made redundant. A constraint directive guarantees that the coupon bond's end time is identical with the number of time steps. We examined the effects of these input parameters on the total price output result.

Example 8.3 (Source file – pkernbw.f90)

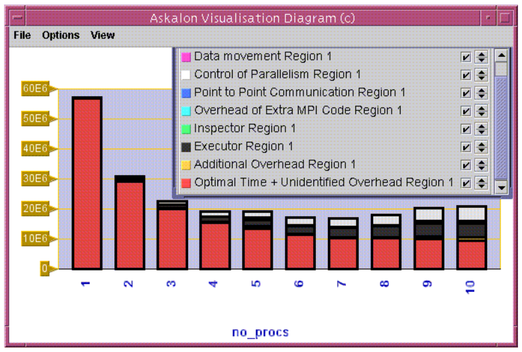
```
read(10,*) nr_steps
!ZEN$ ASSIGN nr_steps = { 5:60:5 }
...
read(10,*) delta_t
!ZEN$ ASSIGN delta_t = { 0.08, 0.17, 0.25, 0.33, 0.42, 0.5, 0.58, 0.67, 0.75, 0.83,
                         0.92, 1 }
...
read(10,*) bond%end
!ZEN$ ASSIGN bond\%end = { 5:60:5 }
!ZEN$ CONSTRAINT VALUE nr_steps == bond\%end
...
read(10,*) bond%coupon
!ZEN$ ASSIGN bond\%coupon = { 0.01:0.1:0.001 }
```

Example 8.4 (Globus RSL Script)

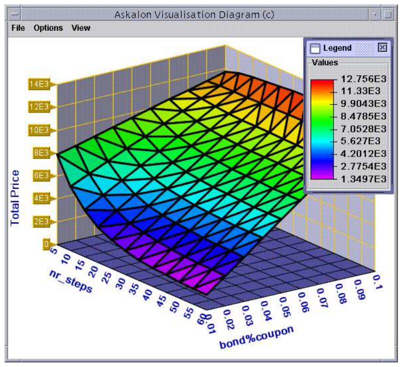
```
+ (&
(*ZEN$ SUBSTITUTE Josie = { Anatevka, Gescher/jobmanager-pbs, Josie }*)
(*ZEN$ CONSTRAINT INDEX Josie == pkernbw.f90:bond\%coupon / 4;
  (resourceManagerContact="Josie")
  (count=4)
  (jobtype=mpi)
  (executable="pkernbw") )
```



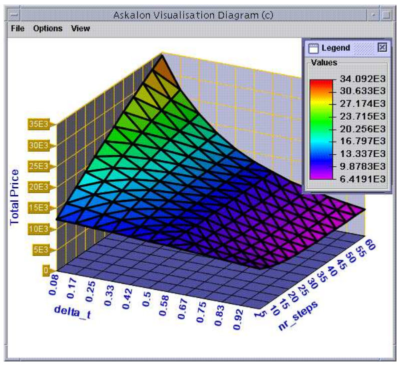
(a) Wallclock time for different number of nodes and OpenMP threads per SMP node.



(b) Contribution of MPI and HPF overheads to the overall execution time (4 threads per SMP node).



(c) Total price for length of time step (delta_t) = 1.0.



(d) Total price for coupon = 0.05.

Figure 8. Visualization Diagrams for Backward Pricing.

Five ZEN directives were inserted into one single source file, which specifies a total of 1481 experiments to be automatically generated and conducted by ZENTURIO. In order to speed-up the completion of this rather large parameter study suite, we annotate the Globus RSL script with three Grid sites on which to schedule the experiments using DUROC: Anatevka, Gescher, and Josie (see Example 8.4). The experiments with $bond\%coupon \leq 0.03$ are scheduled on Anatevka, experiments with $0.04 \leq bond\%coupon \leq 0.07$ are scheduled on Gescher, while experiments with $bond\%coupon \geq 0.08$ are scheduled on Josie. This is expressed by the global index constraint directive illustrated in Example 8.4. By splitting the parameter study onto three Grid sites, we reduced the completion time of the whole suite by more than 50%.

One single Experiment Executor service which runs on the Gescher front-end node has been used to conduct all experiments. Experiments on Anatevka and Josie have been conducted through a Globus

GSI proxy delegation. Upon the completion of each experiment, the standard output which reflects the total price is stored into a single data repository on a separate Grid site.

From the wide variety of visualization diagrams automatically generated during this study, two samples are depicted in Figure 8, which help scientists understand the effect of various inputs on the total price output parameter. A total number of 1481 experiments have been automatically generated. The output of each completed experiment containing the total price has been stored into the data repository. Two sample diagrams that enable the scientists to study the evolution of the total price as a function of the input parameters annotated are depicted in Figure 8.

8.4. Performance Analysis for 3D-Particle-In-Cell with AKSUM

The 3D-Particle-In-Cell [28] is an application written in Fortran90 and MPI simulating the ultrashort laser-plasma interaction in a three dimensional geometry. It can presently run with seven different problem sizes (1, 4, 9, 12, 16, 25 and 36 CPUs).

AKSUM's analysis (Figure 9(a)) shows that the properties *Inefficiency* and *MessagePassingOverhead* are critical in this code. Initially, AKSUM shows the properties ordered by the value of the most severe instance, and the instances of each property organized in a tree manner, with each level showing also the minimum, average, and maximum severity of the property instances under it. Figure 9(a) shows instances of *Inefficiency* organized by number of processes and right below by code regions. RK4_119, MAIN_12, etc. are names AKSUM gave to code regions that were instrumented in the application, while the quadruple before the name is the position (first and last line and column) of the code region. The message "caused by children" indicates that an instance of the same property is present in a child code region with the same or a very close severity value; this fact (most probably) means that the problem is entirely caused by the child code region, and therefore only its children would need to be optimized.

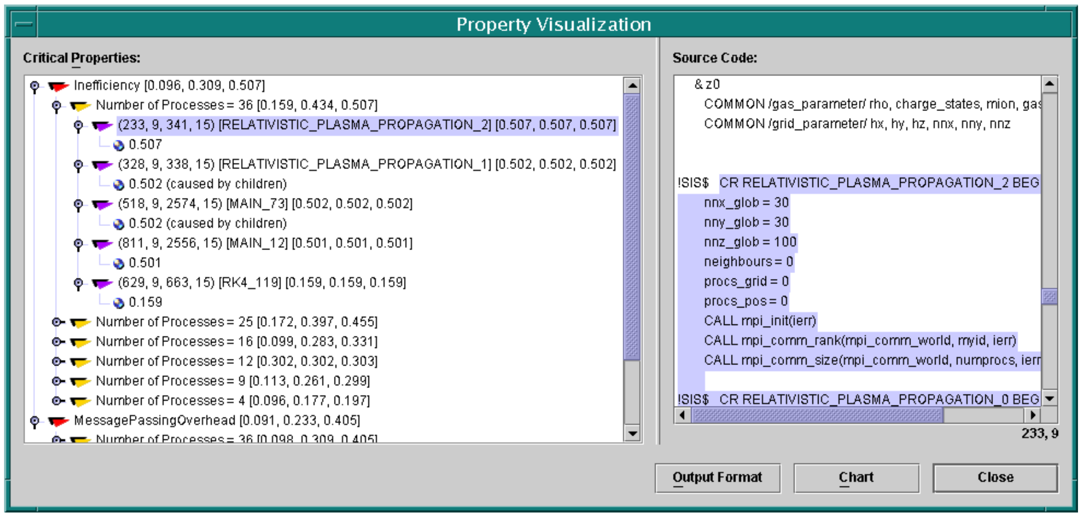
Figure 9(b) shows the instances of *Inefficiency* property plotted in a chart with the help of the ASKALON Visualization Diagrams (described in Section 3.5). The fact that some lines superpose others reflects, again, that instances of *Inefficiency* are present in parent and in child code regions with very close severity values. The chart shows also that the code becomes more inefficient for larger machine sizes, which shows that it does not scale well.

As mentioned in Section 6, properties can be organized in several ways. Figure 9(c) shows how a new organization (first level: code region, second level: property name, third level: number of processes) can be used to find where the most time-consuming MPI call is.

8.5. Modeling and Simulation of a Distributed Scientific Application with PerformanceProphet

The objective of this case study is to examine whether the tool support described in this paper is sufficient to build and evaluate a performance model for a real-world application.

The application for our study LAPW0, which is a part of the WIEN2k package [47], was developed at Institute of Physical and Theoretical Chemistry, Vienna University of Technology. The Linearized Augmented Plane Wave (LAPW) method is among the most accurate methods for performing electronic structure calculations for crystals. The code of LAPW0 application is written by using FORTRAN90 and MPI.


Property Visualization

Critical Properties:

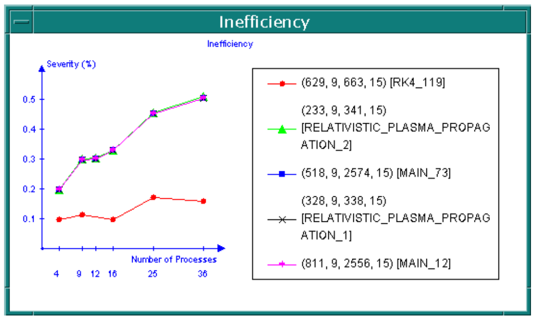
```
Inefficiency [0.096, 0.309, 0.507]
   Number of Processes = 36 [0.159, 0.434, 0.507]
      (233, 9, 341, 15) [RELATIVISTIC_PLASMA_PROPAGATION_2] [0.507, 0.507, 0.507]
         0.507
      (328, 9, 338, 15) [RELATIVISTIC_PLASMA_PROPAGATION_1] [0.502, 0.502, 0.502]
         0.502 (caused by children)
      (518, 9, 2574, 15) [MAIN_73] [0.502, 0.502, 0.502]
         0.502 (caused by children)
      (811, 9, 2556, 15) [MAIN_12] [0.501, 0.501, 0.501]
         0.501
      (629, 9, 663, 15) [RK4_119] [0.159, 0.159, 0.159]
         0.159
   Number of Processes = 25 [0.172, 0.397, 0.455]
   Number of Processes = 16 [0.099, 0.283, 0.331]
   Number of Processes = 12 [0.302, 0.302, 0.303]
   Number of Processes = 9 [0.113, 0.261, 0.299]
   Number of Processes = 4 [0.096, 0.177, 0.197]
MessagePassingOverhead [0.091, 0.233, 0.405]
   Number of Processes = 36 [0.098, 0.309, 0.405]
```

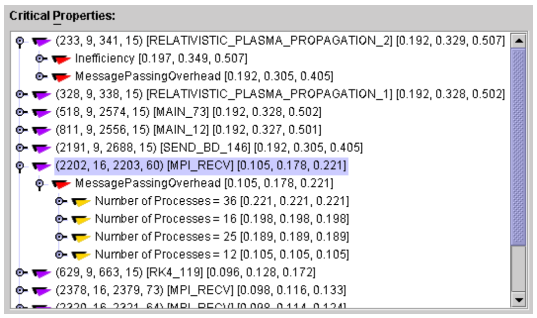
Source Code:

```
  & z0
    COMMON /gas_parameter/ rho, charge_states, mion, gas
    COMMON /grid_parameter/ hx, hy, hz, nnx, nny, nnz


!SIS$  CR RELATIVISTIC_PLASMA_PROPAGATION_2 BEG
    nnx_glob = 30
    nny_glob = 30
    nnz_glob = 100
    neighbours = 0
    procs_grid = 0
    procs_pos = 0
    CALL mpi_init(ierr)
    CALL mpi_comm_rank(mpi_comm_world, myid, ierr)
    CALL mpi_comm_size(mpi_comm_world, numprocs, ierr

!SIS$  CR RELATIVISTIC_PLASMA_PROPAGATION_0 BEG
```

233, 9

(a) Properties found




Critical Properties:

```
(233, 9, 341, 15) [RELATIVISTIC_PLASMA_PROPAGATION_2] [0.192, 0.329, 0.507]
   Inefficiency [0.197, 0.349, 0.507]
   MessagePassingOverhead [0.192, 0.305, 0.405]
(328, 9, 338, 15) [RELATIVISTIC_PLASMA_PROPAGATION_1] [0.192, 0.328, 0.502]
(518, 9, 2574, 15) [MAIN_73] [0.192, 0.328, 0.502]
(811, 9, 2556, 15) [MAIN_12] [0.192, 0.327, 0.501]
(2191, 9, 2688, 15) [SEND_BD_146] [0.192, 0.305, 0.405]
(2202, 16, 2203, 60) [MPI_RECV] [0.105, 0.178, 0.221]
   MessagePassingOverhead [0.105, 0.178, 0.221]
      Number of Processes = 36 [0.221, 0.221, 0.221]
      Number of Processes = 16 [0.198, 0.198, 0.198]
      Number of Processes = 25 [0.189, 0.189, 0.189]
      Number of Processes = 12 [0.105, 0.105, 0.105]
(629, 9, 663, 15) [RK4_119] [0.096, 0.128, 0.172]
(2378, 16, 2379, 73) [MPI_RECV] [0.098, 0.116, 0.133]
(2320, 16, 2321, 64) [MPI_RECV] [0.098, 0.114, 0.124]
```

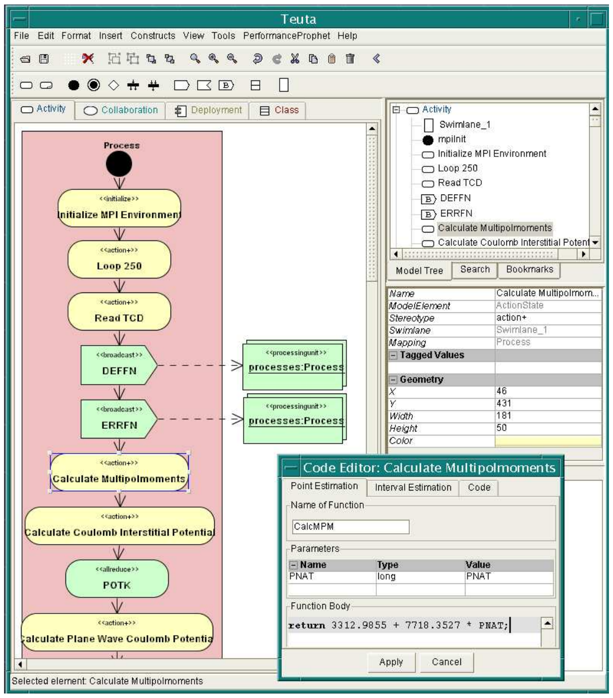
(b) Instances of Inneficiency plotted with one of the ASKALON Visualization Diagrams

(c) Most time-consuming MPI call (MPI_RECV, selected line)

Figure 9. Performance Analysis for 3D-Particle-In-Cell with AKSUM

The LAPW0 application consists of 100 file modules (a module is a file containing source code). The modeling procedure aims to identify the more relevant (from performance point of view) code regions. We call these code regions building blocks. A building block can be a sequence of computation steps, communication operations or input/output operations. In order to assess the execution time of the code regions of LAPW0 application of interest, we have instrumented these code regions and measured their corresponding execution times by using SCALEA [51].

LAPW0 uses two types of MPI collective communication primitives, *broadcast* and *allreduce*. The first one uses a simple binary tree algorithm for implementing the broadcast, whereas the second one is


Teuta
File Edit Format Insert Constructs View Tools PerformanceProphet Help
Activity
Collaboration
Deployment
Class
Process
<<initialize>> Initialize MPI Environment
<<action+>> Loop 250
<<action+>> Read TCD
<<broadcast>> DEFFN
<<broadcast>> ERRFN
<<processingunit>> processes:Process
<<processingunit>> processes:Process
<<action+>> Calculate Multipolmoments
<<action+>> Calculate Coulomb Interstitial Potential
<<allreduce>> POTK
<<action+>> Calculate Plane Wave Coulomb Potential
Activity
Swimlane_1
mpiInit
Initialize MPI Environment
Loop 250
Read TCD
DEFFN
ERRFN
Calculate Multipolmoments
Calculate Coulomb Interstitial Potent
Model Tree
Search
Bookmarks
Point Estimation
Interval Estimation
Code
Apply
Cancel
Selected element: Calculate Multipolmoments

Name	Calculate Multipolmom...
ModelElement	ActionState
Stereotype	action+
Swimlane	Swimlane_1
Mapping	Process
Tagged Values	
Geometry	
X	46
Y	431
Width	181
Height	50
Color	

Code Editor: Calculate Multipolmoments

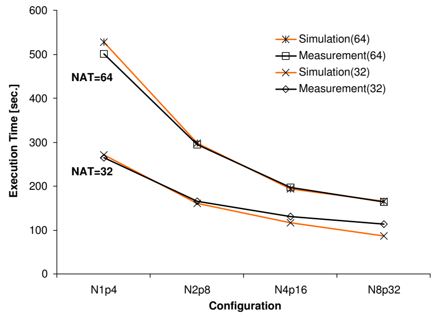
Name of Function: CalcMPM

Parameters

Name	Type	Value
PNAT	long	PNAT

Function Body

```
return 3312.9855 + 7718.3527 * PNAT;
```



(a) The high level UML model

(b) Simulation and measurement results

Figure 10. Modeling and simulation of LAPW0 application with PerformanceProphet

implemented by sending all processes contributions to the root process, and broadcasting from the root the reduced data. Point-to-point MPI communication primitives *receive* and *send* are used for ordering of the output of LAPW0.

Figure 10(a) depicts the model of LAPW0, which is developed with *Teuta*. Due to the size of the LAPW0 model, we can see only a fragment of the UML activity diagram within the drawing space of *Teuta*. On the right hand side of Figure 10(a), is shown how the model of LAPW0 is enriched with cost functions by using *Teuta Code Editor*. A cost function models the execution time of a code region.

In order to evaluate the model of LAPW0, the high-level UML graphical representation of LAPW0 is transformed into the textual representation. *Teuta* automatically generates the corresponding C++ representation, which is used as input for the *Performance Estimator*. The *Performance Estimator* incorporates a parametrised simulator for cluster architectures. The *Performance Estimator* evaluates the performance behavior of LAPW0 application on the user-selected cluster architecture.

Figure 10(b) shows the simulation and measurement results for two problem sizes and four machine sizes. The problem size is determined by the parameter NAT, which represents the number of atoms in a unit of the material. The machine size is determined by the number of nodes of the cluster architecture. Each node of the cluster has four CPU's. One process of the LAPW0 application is mapped to one CPU of the cluster architecture. The performance model is validated by comparing the simulation results with the measurement results. We consider that this performance model provides the performance prediction results with the accuracy which would be sufficient to compare various designs of the LAPW0 application.

9. Conclusions

The development of the ASKALON tool set has been driven by the need of scientists and engineers to perform performance analysis, experiment management, parameter studies, modeling, and prediction of parallel and distributed applications for cluster and Grid infrastructures. ASKALON supports these functionalities through the provision of four sophisticated tools: SCALEA for instrumentation and performance analysis, ZENTURIO for experiment management, performance and parameter studies, AKSUM for automatic bottleneck detection and performance interpretation, and PerformanceProphet for performance modeling and prediction. Each tool can be accessed and manipulated via advanced User Portals. ASKALON has been designed as a distributed Grid service-based architecture and implemented on top of the OGSI technology and Globus toolkit. Designing each tool as a composition of remote Grid service provides a series of advantages: (1) isolates platform dependencies on specific critical sites under a well-defined portable API; (2) enables light-weight clients, easy to be installed and managed by users on local sites (e.g. on notebooks); (3) allows the interaction of multiple tools by accessing resources concurrently through common shared services. The ASKALON tools exchange information through a common Data Repository or interoperate through the underlying Grid services. A generic visualization package that supports a wide variety of portable diagrams in both post-mortem and on-line modes is employed by the User Portals of all tools.

Currently, we are working on a more coherent integration and interoperability of all tools to reflect the continuously evolving Globus, OGSI, and Web service-based Grid specifications. In addition, each tool will be extended with new functionality. SCALEA will be enhanced with more advanced Grid application monitoring and analysis. ZENTURIO will be extended with a generic application optimization framework for Grid application scheduling, in particular for large workflows. Application prediction information crucial for good scheduling will be provided by PerformanceProphet through the Data Repository. AKSUM is currently being enhanced for automatic performance analysis of Java applications based on the JavaSymphony [21] programming model for the Grid. The PerformanceProphet technology will combine software engineering with performance modeling and analysis.

REFERENCES

1. D. Abramson, R. Sosic, R. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations high performance parametric modeling with nimrod/G: Killer application for the global grid? In *Proceedings of the 4th IEEE Symposium on High Performance Distributed Computing (HPDC-95)*, pages 520–528, Virginia, August 1995. IEEE Computer Society Press.
2. V. Adve, R. Bagrodia, J. Browne, E. Deelman, A. Dube, E. Houstis, J. Rice, R. Sakellariou, D. Sundaram-Stukel, P. Teller, and M. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Transactions on Software Engineering*, 26:1027–1048, November 2000.
3. 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 Computing*, 28(5):749–771, May 2002.

4. APART – IST Working Group on Automatic Performance Analysis: Real Tools, Aug 2001 until July 2004.
5. Bob Atkinson, Giovanni Della-Libera, Satoshi Hada, Maryann Hondo, Phillip Hallam-Baker, Johannes Klein, Brian LaMacchia, Paul Leach, John Manferdelli, Hiroshi Maruyama, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, John Shewchuk, and Dan Simon. Web Services Security (WS-Security). Specification, Microsoft Corporation, April 2002.
6. S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming, IOS Press, The Netherlands*, 7(1):67–81, 1999.
7. 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.
8. Nat Brown and Charlie Kindel. *Distributed Component Object Model protocol: DCOM/1.0*. Microsoft Corporation and Redmond, WA, January 1998.
9. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceeding SC'2000*, November 2000.
10. 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.
11. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL), March 2001. http://www.w3.org/TR/wsdl.
12. Michel Courson, Alan Mink, Guillaume Marcais, and Benjamin Traverse. An automated benchmarking toolset. In *HPCN Europe*, pages 497–506, 2000.
13. Francisco Curbera, David Ehnebuske, and Dan Rogers. Using WSDL in a UDDI Registry 1.07. UDDI best practice, UDDI Organisation, May 2002. http://www.uddi.org/pubs/wsdlbestpractices-V1.07-Open-20020521.pdf.
14. Karl Czajkowski, Ian Foster, Nick Karonis, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.
15. Karl Czajkowski, Ian Foster, and Carl Kesselman. Co-allocation services for computational grids. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1999.
16. 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.
17. W. K. Edwards. Core Jini. *IEEE Micro*, 19(5):10–10, September/October 1999.
18. T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, and H.-L. Truong. ASKALON - A Programming Environment and Tool Set for Cluster and Grid Computing. www.par.univie.ac.at/project/askalon, Institute for Software Science, University of Vienna.
19. 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.
20. Thomas Fahringer. ASKALON Visualization Diagrams. http://www.par.univie.ac.at/project/askalon/visualization/index.html.
21. Thomas Fahringer and Alexandru Jugravu. JavaSymphony: New Directives to Control and Synchronize Locality, Parallelism, and Load Balancing for Cluster and GRID-Computing. In *ACM Java Grande - ISCOPE 2002 Conference*, Seattle, November 2002. ACM.
22. Thomas Fahringer and Clovis Seragiotto. Modeling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. In *Proceeding SC'2001, Denver, USA*, November 2001.
23. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
24. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. The Globus Project and The Global Grid Forum, November 2002. http://www.globus.org/research/papers/OGSA.pdf.
25. 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.
26. 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.
27. Apache Software Foundation. Apache Axis. http://ws.apache.org/axis.
28. M. Geissler. *Interaction of High Intensity Ultrashort Laser Pulses with Plasmas*. PhD thesis, Vienna University of Technology, 2001.
29. William Grosso. *Java RMI*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 2002. Designing and building distributed applications.
30. Elliotte Rusty Harold. *XML: EXtensible Markup Language*. IDG Books, San Mateo, CA, USA, 1998.

31. Rolf Herzog. PostgreSQL — the Linux of databases. *Linux Journal*, 46:??–??, February 1998.
32. Yannis E. Ioannidis, Miron Livny, S. Gupta, and Nagavamsi Ponnekanti. ZOO: A desktop experiment management environment. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, pages 274–285, Mumbai (Bombay), India, 3–6 September 1996. Morgan Kaufmann.
33. Karen L. Karavanic and Barton P. Miller. Experiment management support for performance tuning. In ACM, editor, *Proceedings of the SC'97 Conference*, San Jose, California, USA, November 1997. ACM Press and IEEE Computer Society Press.
34. G. Krasner and S. Pope. A cookbook for using the Model-View-Controller interface paradigm. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.
35. David S. Linthicum. CORBA 2.0? *Open Computing*, 12(2), February 1995.
36. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, Washington, D.C., USA, June 1988. IEEE Computer Society Press.
37. 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.
38. 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*, 1995.
39. OMG. Unified Modeling Language Specification. http://www.omg.org, March 2003.
40. S. Pllana and T. Fahringer. On Customizing the UML for Modeling Performance-Oriented Applications. In *<<UML>> 2002, "Model Engineering, Concepts and Tools", LNCS 2460, Dresden, Germany*. Springer-Verlag, October 2002.
41. 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.
42. Radu Prodan and Thomas Fahringer. ZEN: A Directive-based Language for Automatic Experiment Management of Parallel and Distributed Programs. In *Proceedings of the 31st International Conference on Parallel Processing (ICPP-02)*, Vancouver, Canada, August 2002. IEEE Computer Society Press.
43. Radu Prodan and Thomas Fahringer. ZENTURIO: A Grid Middleware-based Tool for Experiment Management of Parallel and Distributed Applications. *Journal of Parallel and Distributed Computing*, 2003. To appear in Special Issue on Middleware.
44. 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.
45. Bill Roth. An introduction to Enterprise Java Beans technology. *Java Report: The Source for Java Development*, 3, October 1998.
46. A. Ryman. Simple Object Access Protocol (SOAP) and Web Services. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 689–689, Los Alamitos, California, May12–19 2001. IEEE Computer Society.
47. K. Schwarz, P. Blaha, and G. Madsen. Electronic structure calculations of solids using the WIEN2k package for material sciences. *Computer Physics Communications*, 147:71–76, 2002.
48. H. Schwetman. Hybrid Simulation Models of Computer Systems. *Communications of the ACM*, 21(9):718–723, 1978.
49. H.M. Stommel. The western intensification of wind-driven ocean currents. *Transactions American Geophysical Union*, 29:202–206, 1948.
50. Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany. *A Grid Monitoring Architecture*. The Global Grid Forum, January 2002. http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-16-2.pdf+.
51. Hong-Linh Truong and Thomas Fahringer. SCALEA: A Performance Analysis Tool for Distributed and Parallel Program. In *8th International Europar Conference(EuroPar 2002)*, Lecture Notes in Computer Science, Paderborn, Germany, August 2002. Springer-Verlag.
52. Hong-Linh Truong and Thomas Fahringer. On Utilizing Experiment Data Repository for Performance Analysis of Parallel Applications. In *9th International Europar Conference(EuroPar 2003)*, Lecture Notes in Computer Science, Klagenfurt, Austria, August 2003. Springer-Verlag.
53. 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.
54. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. *Grid Service Specification*. The Globus Project and The Global Grid Forum, February 2002. http://www.globus.org/research/papers/gsspec.pdf.
55. UDDI: Universal Description, Discovery and Integration. http://www.uddi.org.
56. W3C. Web Services Activity. http://www.w3.org/2002/ws/.

57. R. Wismüller and T. Ludwig. THE TOOL-SET – An Integrated Tool Environment for PVM. In H. Lidell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *Proc. High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 1029–1030, Brussels, Belgium, April 1996. Springer-Verlag.
58. R. Wismüller, J. Trinitis, and T. Ludwig. OCM — A Monitoring System for Interoperable Tools. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*, Welches, OR, USA, August 1998. ACM Press.
59. 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.
60. M. Yarrow, K. M. McCann, R. Biswas, and R. F. Van der Wijngaart. Ilab: An advanced user interface approach for complex parameter study process specification on the information power grid. In *Proceedings of Grid 2000: International Workshop on Grid Computing*, Bangalore, India, December 2000. ACM Press and IEEE Computer Society Press.