ZENTURIO: An Experiment Management System for Cluster and Grid Computing*

Radu Prodan and Thomas Fahringer
Institute for Software Science
University of Vienna
Liechtensteinstrasse 22
A-1090 Vienna, Austria
{radu,tf}@par.univie.ac.at

ABSTRACT

Over the last decade, a dramatic increase has been observed in the ability to generate and store data in the course of parameter studies, performance analysis, and software testing. In this paper, we introduce ZENTURIO, a sophisticated experiment management environment that greatly alleviates parameter studies, performance analysis, and software testing for a wide variety of target architectures, including cluster and Grid architectures. ZENTURIO uses a novel directive-based language named ZEN in combination with a user portal to specify arbitrary complex program executions. Among others, ZEN enables to substitute strings, to request for performance information, and to control the number of experiments. ZENTURIO has been designed as a collection of Grid-services which comprise: (1) a registry service which supports registering and locating Grid services; (2) an experiment generator that parses files with ZEN directives and instruments applications for performance analysis and parameter studies; (3) an experiment manager that compiles and controls the execution of experiments on the target machine. A graphical user portal allows the user to control and monitor the experiments and to automatically visualize performance and output data across multiple experiments. ZENTURIO has been implemented by employing Java/Jini technologies and supports experiments on cluster architectures via PBS and on Grid infrastructures through Globus schedulers. We report results of using ZENTURIO with an actual performance analysis and parameter study for a Globus site and a cluster architecture, respectively.

Keywords: experiment management, parameter studies, performance analysis, user portal, Jini, cluster, Grid.

1. INTRODUCTION

The development and execution management of scientific and engineering applications on complex, heterogeneous and non-dedicated distributed architectures, ranging from cluster architectures [2] to widely distributed Grid structures [14], is a tedious, time-consuming and error-prone process.

On the one hand, efficient parameter studies have become feasible through the appearance of parallel compute engines with multi-gigabyte memories and terabyte disk farms. Scientists and engineers perform parameter studies for large applications to obtain solution information for a wide variety of input parameter values. On the other hand, the development of performance-oriented applications involves many cycles of code editing, performance tuning, code execution, testing, and performance analysis. Some performance metrics, such as scalability or speedup, may require the testing of numerous problem and machine sizes for different compiler options and target architectures.

To this date, still little attention has been devoted to the needs of scientists and engineers who manually create their parameter studies, manage many different sets of input data, launch large number of program compilations and executions, administer corresponding result files, invoke performance analysis tools to derive performance metrics, relate performance data back to experiments (execution of a program) and code regions, etc.

This paper describes ZENTURIO, a novel automatic experiment management framework for cluster and Grid architectures. ZENTURIO uses a directive-based language called ZEN [19] to specify arbitrarily complex program executions in the context of performance analysis and tuning, parameter studies, and software testing. ZEN enables the programmer to invoke experiments for arbitrary value ranges of any problem parameter, including program variables, file names, compiler options, target machines, machine sizes, scheduling strategies, data distributions, etc. Moreover, ZEN directives can be used to request for a large variety of performance metrics (e.g. cache misses, load imbalance, execution, communication, synchronisation time, etc.) for arbitrary code regions (e.g. entire program, procedures, loops, statements, etc.). ZEN directives are interpreted as comments and ignored by systems that are unaware of their semantics. ZENTURIO parses application files annotated with ZEN directives. Based on the semantics of the directives encountered it generates appropriate application codes. Source code as part of application files can be optionally instrumented through an interface to an external instrumentation engine. The instrumented applications are transferred to the target machine for compilation and execution. Upon their completion, the output files and performance data are stored into a data repository for multi-experiment performance and parameter studies.

We have fully implemented ZENTURIO as a distributed Grid-based framework which is based on three generic Grid services implemented by Java/Jini technology. (1) The Registry Service supports registration and location of Grid services; (2) The Experiment Generator generates the set of desired experiments based on the ZEN directives inserted in the source code. (3) The Experiment Executor (Section 4.3) compiles, executes, and monitors experiments on a target architecture. Interfaces are provided for applications running on cluster architectures via PBS (Portable Batch

*This research is supported by the Austrian Science Fund as part of the Aurora project under contract SFBF1104.

System [27]) and on Grid infrastructures via the Globus metacomputing environment [10]. A graphical User Portal (Section 4.2) enables the user to control and monitor all experiments.

The rest of this paper is organised as follows: The next section discusses related work. Section 3 presents an overview of the ZEN language. Section 4 describes the design principles of ZENTURIO, followed by a section about the implementation strategy. Results of a performance and parameter study are reported in the next section. Finally, some concluding remarks are made and future work is outlined in Section 7.

2. RELATED WORK

Our work on a system design for automatic experiment management is centered around four different research areas: scientific experiment management, performance analysis, parameter studies, and Grid service architectures.

The Paradyn performance analysis tool [16] supports performance experiment management through: (1) techniques for quantitatively comparing several experiments, and (2) performance diagnosis based on dynamic instrumentation. Experiments have to be set up manually, whereas performance analysis is done automatically for every different experiment. Paradyn's analysis focuses on functions and function calls, whereas ZEN supports performance analysis for arbitrary code regions.

The ZOO project [15] has been initiated to support scientific experiment management by providing an experiment specification language and by supporting automatic experiment execution and data exploration. Parameterisation is limited to input and output files.

Nimrod [1] is a tool that manages the execution of parameter studies. A parameterised experiment is specified by a declarative plan file. Nimrod generates one job for each unique combination of parameter values, by taking the cartesian product of all values. Unlike in ZENTURIO, the set of possible parameter value combinations cannot be restricted. String substitution in program files is not allowed. Moreover, Nimrod can only manage already compiled applications and parameters outside of source files.

The ILAB [28] project controls parameter studies through a graphical user interface for input files. Several job execution models are provided for interacting with different schedulers, for instance, PBS and Globus schedulers.

The Unicore project [20] facilitates the usage of supercomputers on the Internet. Unicore jobs are organised manually as a directed acyclic graph, consisting of job groups and tasks. Such jobs comprise file transfers and binary/script file executions, including compilation and linking tasks. There is no support for automatic invocation of multiple experiments and performance analysis.

Recently, an Open Grid Services Architecture (OGSA) specification [11, 25] has been jointly developed by the Open Grid Service Infrastructure (OGSI) working group (part of the Global Grid Forum) and members of the GLOBUS [10] project. The design and implementation of ZENTURIO has been started well before the first OGSA draft documents appeared. However, OGSA and ZENTURIO share many common features. Future development of ZENTURIO will focus on a more close integration of ZENTURIO with OGSA, in particular by employing Web Services and the underlying XML-based technologies [17].

3. ZEN LANGUAGE

In order to support the user in the specification of a large number of experiments under ZENTURIO, we defined a directive-based language called ZEN, described in [19]. ZEN supports users in specifying arbitrarily complex program executions, to control and guide parameter studies, performance analysis, and software testing for a wide variety of parallel and distributed architectures. ZEN directives are comment lines starting with the `!ZEN$` prefix and, therefore, ignored by systems that are unaware of their semantics. Apart from the special characters which mark the beginning (and eventually the end) of a comment, ZEN directives are language independent. Such an approach allows to specify arbitrary value ranges of any problem, system, or machine parameter, including program variables, file names, compiler options, target machines, machine sizes, scheduling strategies, data distributions, etc. Moreover, ZEN directives can be restricted to arbitrary code regions of any file.

Example 3.1 (Sample ZEN file)

```
1: !ZEN$ CR CR_P PERF ODATA, WTIME
2: !ZEN$ ASSIGN P = { {8,16}**2 }
3: !ZEN$ SUBSTITUTE STATIC = { STATIC, DYNAMIC, GUIDED }
4: !ZEN$ CONSTRAINT INDEX 4*(NUM_THREADS\(4\) - 1) < D &&
                          D <= 4 * NUM_THREADS\(4\) BEGIN
5:   !ZEN$ SUBSTITUTE NUM_THREADS\(4\) =
                           { NUM_THREADS\({4,8}\} } BEGIN
6:     !ZEN$ ASSIGN D = { 2**{6:12} }
7:     !ZEN$ CR CR_OMPPA PERF OSYNC, ODATA_L21 BEGIN
8:       !$OMP PARALLEL DO SCHEDULE(STATIC) NUM_THREADS(4)
9:         DO i = 1, D
            ...
10:        ENDDO
11:  !ZEN$ END SUBSTITUTE
12:    !ZEN$ END CR
13: !ZEN$ END CONSTRAINT
14: !ZEN$ CONSTRAINT VALUE D^3 / P < 10000000
```

Example 3.2 (1 Sample ZEN file instance for Ex. 3.1)

```
!SIS$ CR CR_P PERF ODATA, WTIME
P = 8**2
D = 2**6
!SIS$ CR CR_OMPPA PERF OSYNC, ODATA_L21 BEGIN
    !OMP$ PARALLEL DO SCHEDULE(STATIC) NUM_THREADS(4)
        DO i = 1, D
           ...
        ENDDO
!SIS$ END CR
```

Example 3.1 shows an excerpt of a Fortran OpenMP file annotated with ZEN directives. The ZEN language defines four types of directives: substitute, assignment, constraint, and performance behaviour. A variable assigned with a set of values (also called *value set*) within a ZEN substitute or assignment directive is called *ZEN variable* (e.g. `P`, `STATIC`, `NUM_THREADS(4)`, and `D`). Large value sets assigned to ZEN variables can be expressed in a very compact form through the usage of ZEN sets, whose syntax is specified in [19]. Additionally, value sets are defined as totally ordered sets, where the ordering operation is well defined by the ZEN set specification. ZEN variables can be of type *integer*, *real*, or *string*. The type is determined during the ZEN language compilation phase, based on the value set elements.

The *substitute directive* causes all occurrences of the ZEN variable name within the directive's scope be replaced by one element from its value set. ZEN substitute directives can have a global scope in the entire ZEN file (see directive `3` in Ex. 3.1), or local scope (directive `5` in Ex. 3.1, between statements `5` to `11`). Directive `5` sets the number of threads to 4, respectively 8, for all OpenMP parallel loops in its scope. Directive `3` defines the set of scheduling strategies to be examined for the all OpenMP parallel loops in the program.

The directive based approach allows the user to annotate ZEN files without altering the semantics of the source code.

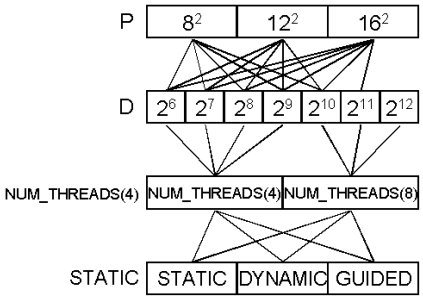


Figure 1: The constraints induced by the ZEN constraint directives from Ex. 3.1.

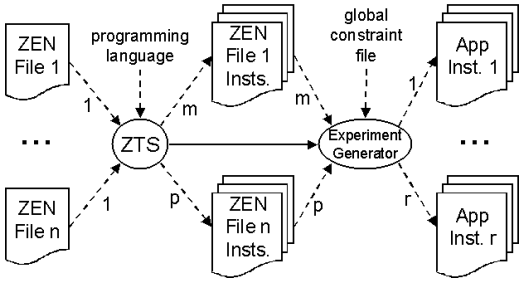


Figure 2: The ZEN Application Instance generation.

However, the substitute directive should not be used to indicate all values of interest for variable `D` in statement `9`, as it would replace all occurrences of this character in this statement, including in the keyword `DO`. Instead of forcing the user to change variable names to overcome this limitation, we introduce the assignment directive.

The *assignment directive* (directives `2` and `6` in Ex. 3.1) implies that an assignment statement is inserted in the program, which assigns one ZEN variable (which must be a valid program variable) with one element from its value set. The assignment statement textually replaces the directive in the generated code. In Ex. 3.1, directive `2` sets the number of processors to the values 8^2 and 16^2, and directive `6` assigns the values 2^6,2^7,...,2^{11},2^{12} to variable D.

By default, the cartesian product of the value sets of all ZEN variables defined is computed, which would result in the same number of experiments. In order to limit this to a smaller number of valid combinations the ZEN *constraint directive* is introduced. It defines a boolean expression with a well defined syntax [19] over a set of ZEN variables. The effect is to constrain the value sets of the ZEN variables involved to a meaningful subset of combinations, for which the boolean expression yields true. In order to cope with the different ZEN variable types and provide more flexibility to the user, two types of constraint directives are defined: (1) *value constraints* apply directly to the value set elements (useful for integers and reals); (2) *index-domain constraints* apply to the element indexes in the totally ordered value sets (useful for strings). Constraint directives can have global scope in the entire file (directive `14`) or local scope (directive `4` with scope ranging from `4` to `13`). Directive `4` constrains the number of threads to execute the parallel region, which depends on the problem size D. Directive `14` restricts the combinations of machine and problem sizes (`D` and `P`) to those which need less than 40MB on each processor (see Fig. 1). Global constraints over ZEN variables in different ZEN files can be indicated in a *global constraint file*. In this case ZEN variable must be prefixed by the ZEN file name followed by a colon (see Ex. 6.6 in Section 6).

The *performance behaviour directive* defines performance metrics to be measured for specific code regions. Overall, ZEN supports approximately 50 code regions (e.g. entire program, all OpenMP parallel loops, all MPI code regions, etc.) and 40 performance metric mnemonics (e.g. data movement, and synchronisation) for various parallel programming paradigms, including OpenMP [6], HPF [13], and MPI [18]. In our example, directive `1` measures the global wall-clock time and data movement (i.e. communication) for the entire program unit. Directive `7` measures the synchronisation overhead and the number of cache misses in the OpenMP parallel loop.

Local ZEN directives can be nested or interleaved. For instance, the constraint directive with the scope between `4` and `13` encloses the substitute directive with the scope between `5` and `11`. The substitute directive with the scope between `5` and `11` interleaves with the performance behaviour directive with the scope between `7` and `12`.

A file annotated with ZEN directives is called a *ZEN file*. A *ZEN Transformation System (ZTS)* parses a ZEN file and generates a set of *ZEN file instances*. A ZEN file instance instantiates each ZEN variable with one element from its value set, based on the semantics of the ZEN directives. Assignment statements, which textually replace the ZEN assignment directives, are generated using the syntax of the programming language given as input to ZTS (see Fig. 2). The ZEN performance behaviour directives are translated to instrumentation directives or command-line options of the selected external instrumentation engine. For the ZEN file in Ex. 3.1, based on the constraints depicted in Fig. 1, a total of 48 ZEN file instances will be generated by ZTS. Example 3.2 illustrates a single ZEN file instance of the ZEN file shown in Ex. 3.1. An application, consisting of a set of files, some of which being annotated with ZEN directives, is called *ZEN application*. A *ZEN application instance* instantiates each ZEN file with one ZEN file instance. The set of application instances generated for a ZEN application is given by the cartesian product of the sets of ZEN file instances created for each constituent ZEN file. A global constraint file filters the application instances to a subset that fulfills all constraints (see Fig. 2).

An *experiment* is an entity, which is fully specified by a ZEN application instance, a compilation and execution command, a compilation and an execution directory, and a set of output file names. Experiments are executed on a target machine and produce a set of output files and performance data.

A detailed description of ZEN can be found in [19].

4. ZENTURIO

ZENTURIO is a system which enables the user to automatically conduct a large number of experiments (up to thousands) on cluster and Grid architectures. Its goal is to support parameter studies, software testing, and performance analysis and tuning. ZENTURIO is based on ZEN (see Section 3), which is a directive-based language for annotating arbitrary files. It specifies value ranges for any problem, system, or machine parameter, including program variables, file names, compiler options, target machines, machine sizes, scheduling strategies, data distributions, etc. Currently ZEN directives must be manually inserted in ZEN files. ZENTURIO receives as input a ZEN application, including the application source and input files, the set of output files, one compilation and one execution command, and the machine name on which to execute the experiments of interest. ZENTURIO automatically instruments the ZEN

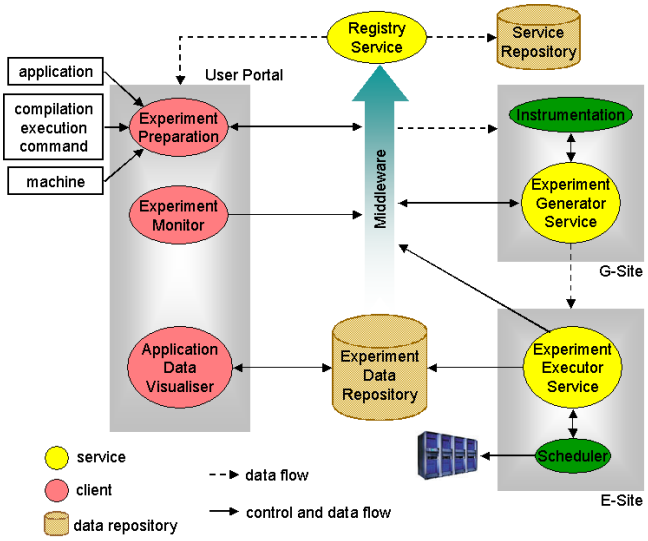


Figure 3: ZENTURIO basic architecture.

application by incorporating ZTS (see Fig. 2) and generates all experiments. Thereafter, it automatically transfers, compiles, and runs the experiments on the target machine (see Section 4.3). Transferring and compiling applications are optional tasks that can be omitted, for instance, when the client (site of user) and the target machines share a file system and the application has already been compiled.

After an experiment has completed, the results (output files and performance data) can be optionally stored into a data repository. A GUI-based portal (see Section 4.2) enables the user to on-line monitor and control experiments and to visualise the results after they have completed.

4.1 Architecture Design

ZENTURIO is designed as an open Grid architecture, depicted in Fig. 3, which consists of several Grid services distributed over several Grid sites. A *Grid site* is a host that can be accessed over the Internet. A *Grid service* is an application that runs on a Grid site and can be remotely invoked through a well-defined interface.

All services register with a *Registry Service (RS)*, thus making themselves publicly available. The RS stores all registered services including itself in a *Service Repository* such as JavaSpaces [12], UDDI [26], or LDAP [7]. The repository can be either centralised, as shown in Fig. 3, or distributed over multiple sites. RS grants *leases* to services for a certain period of time. If a service does not renew its lease before it has expired, then the RS will delete it. Leasing is a very effective and efficient way to cope with network failures and to ensure network resilience. A *notification* mechanism is used to inform the clients (i.e. the User Portal) about new services that registered with the RS and about existing services that failed to renew their lease. Thereby, clients are always provided with a dynamically updated view of the Grid service environment.

In order to become a Grid service, an application must implement a common *Grid Service Interface (GSI)*. This interface specifies common functionality provided by every Grid service and includes methods for retrieving the service address, acquiring detailed service information which is not stored in the RS, terminating the service, specifying the lifetime of a service (upon its termination the service automatically stops itself), renewing the lease, retrieving the load of a service, and subscribing for and launching notification callbacks (e.g. in case of system faults).

The *User Portal (UP)* (see Section 4.2) is a GUI-based client, which represents the entry point of the system for a user. It normally resides on the user's local machine. Its main purpose is to load ZEN applications, submit, control, and monitor experiments on the target machine, and to visualise performance and output data.

ZENTURIO provides three distinct Grid services, for which arbitrarily many instances distributed over several Grid sites may exist. The *Registry Service* registers existing services and supports clients to locate services of interest. The *Experiment Executor (EE)* service (see Section 4.3) manages experiments on the target machine, called *E-site*. The EE enables compilation and control of experiments. Upon completion, it offers to store the experiment's output and performance data into the *Experiment Data Repository*. The *Experiment Generator (EG)* service resides on the *G-site* (which could coincide with the E-site) and uses ZTS to instrument ZEN files and to generate appropriate application instances. Thereafter, it associates each application instance with a compilation command, an execution command, and a target machine name on which the application will be executed. The experiments are generated on the local G-site, each one in a separate directory. Experiments can be copied to a target E-site or deleted if no longer needed.

ZENTURIO also supports performance instrumentation of parallel and distributed programs. For this purpose, the ZENTURIO instrumentation interface has been integrated with the SCALEA instrumentation engine [24], which provides a full Fortran90 OpenMP/MPI/HPF front-end and unparser. ZENTURIO translates the ZEN performance behaviour directives to SCALEA directives and command-line options. In future work we will integrate other performance instrumentation systems that enable instrumentation of C, C++, and Java codes as well. All other ZEN directives are parsed by the language independent ZTS.

In order to encapsulate the instrumentation system from the rest of ZENTURIO, we decided to separate the EG (including the instrumentation engine) from EE. Therefore, for any new target machine that is accessible via ZENTURIO, EG does not have to be reinstalled. By running EG and EE as a publicly available services, users can dynamically connect to these services and generate/execute their experiments on sites where they have been installed. By doing so, the tedious task of installing and managing software is substantially reduced.

All methods of EE and EG are provided in synchronous (blocking) and asynchronous (non-blocking) mode. *Asynchronous methods* can optionally invoke responses via *notification* callbacks, which enable highly responsive clients. This functionality is particularly useful for the Experiment Monitor as it avoids expensive polling for information. Furthermore, the EE and EG services can be *accessed concurrently* by multiple clients, which is yet another key functionality for scalable Grid infrastructures.

The *Experiment Data Repository* is used to store information about the application and all experiments, including output and performance data. The user can access this data via the UP for post-mortem performance analysis and visualisation. The Experiment Data Repository can be concurrently accessed by multiple sites (i.e. multiple EEs) and tools (e.g. visualisation tools).

4.2 User Portal (UP)

The UP is the client part of ZENTURIO. It provides the user with a GUI-based interface for submitting, monitoring, controlling, and analysing experiments for Grid and cluster

architectures. The UP consists of three client components: *Experiment Preparation (EP)* client, *Experiment Monitor (EM)* client, and *Application Data Visualiser (ADV)* client.

4.2.1 Experiment Preparation (EP)

The EP is responsible of preparing a set of experiments, based on the following inputs provided by the user: (1) a ZEN application, given by a list of source, script, and input files which can be individually selected from arbitrary directories; (2) the compilation directory and the compilation command; (3) the execution directory and the execution command; (4) the machine on which the experiments should be executed (i.e. the E-site); (5) the scheduler to be used; (6) the output files/directories.

The application files are categorised into ZEN files (to be processed by ZTS) and regular files (without any ZEN directives). ZEN files are split into ZEN source files and ZEN script/input files. ZEN source files require compilation of the generated ZEN application instances whereas ZEN script/input files do not. ZTS generates application instances for ZEN variables that appear in ZEN script/input files such that they share the ZEN source file instances without recompilation.

After all inputs have been received by the EP, it automatically contacts the RS for an available EG service (preferably on the local site or on the E-site to minimise file transfers) in order to generate the corresponding experiments. If the EG resides on a remote site, the ZEN application with all its files is transferred to the G-site (via `ftp`). After the experiments have been generated by the EG, they can be submitted to the EE for compilation and execution. If a machine name is provided by the user, then an EE running on this machine is retrieved from the RS. If no machine is specified, then a default machine is selected by ZENTURIO. If the E-site is different from the G-site, then the EP client requests the EG to transfer all experiments to the E-site. Although ZENTURIO reports any compilation errors to the user via the UP, it is recommended that applications are implemented as self-contained programs that reference only relative paths to other files.

The UP can also reload applications whose experiments (jobs) have been previously submitted for execution. A user may decide to close a UP (e.g. at night time) while the jobs are still being processed. At a later point in time, the user may start a new UP, verify the status of his experiments and visualise and analyse the results. Any kind of compiler or runtime error will be stored and forwarded to the UP. Based on this information the user can proceed with his study (e.g. change problem parameters, debug application, etc.).

4.2.2 Experiment Monitor (EM)

The EM uses the EE service to compile, execute, control, and monitor experiments on-line. The user interface for the EM is shown in the right panel of the UP in Fig. 4. By selecting one ZEN application in the EP (left) panel, the corresponding set of experiments are displayed by the EM. All experiments of a specific application can be submitted for execution by selecting the corresponding ZEN application and clicking on a single button. As part of the EM, an *event listener* (thread) receives notifications from the EE about changes in the status of all experiments, thus avoiding expensive polling. These changes are displayed in the UP while the experiments are being executed. By clicking on a single experiment, all ZEN variables that describe the experiment with their associated values are shown. The ZEN variables are grouped with respect to the ZEN files in which they occur. For instance, Fig. 4 displays the values of the ZEN variables *points* and *iters* which are included in the file *st.in* of an experiment (uniquely identified with the number 25) whose status is COMPILING.

Figure 5 represents the experiment state transition diagram which shows every state an experiment can assume during its lifetime.

4.2.3 Application Data Visualiser (ADV)

ZENTURIO defines two types of output results for an experiment. The performance metrics (e.g. execution time, synchronisation, communication) are computed based on the ZEN behaviour directives. The output files are indicated by the user in the experiment preparation phase. Upon experiment termination, both performance metrics and output files are stored into the *E*xperiment Data Repository. The *Application Data Visualiser (ADV)* can be employed post-mortem by the user to automatically visualise performance and output data across a certain subset of experiments that have been executed for a ZEN application. The types of visualisation supported include line-chart, bar-chart, pie-chart, and terrain.

Figure 6 shows the ADV dialog box for constructing a multi-experiment visualisation. The top left panel shows various performance metrics (e.g. barrier, collective communication, control of parallelism) which can be selected for visualisation. There are two views provided by ZENTURIO: (1) *Metric-to-Region* (shown in Fig. 6) displays the list of metrics computed. A set of regions is tabulated for which this metric was greater than zero; (2) *Region-to-Metric* (not shown in the paper) displays a hierarchy of code regions, for which a computed metric can be selected. Single or multiple metric selection is allowed. The top-right panel always displays the code region corresponding to the last selected metric. The metric value constituted one axis in the resulting visualisation diagram. In case of a multiple selection, the axis onto which the metrics are varied can be selected in the menu. The bottom panel displays all ZEN variables of the current ZEN application. Every ZEN variable has an associated list box that shows its value set. The visualisation will be generated for all experiments whose ZEN variables match the selected values. We have defined two special values for ZEN variable values:

WildCard matches any ZEN variable value. The corresponding ZEN variable is selected as a visualisation axis. The elements of its value set are sorted alphabetically and displayed on the axis. For the moment a maximum of three wild cards is allowed. One, two, or three wild card selections, respectively imply a 2D, 3D, or 4D visualisation. To control the axis onto which the values of a ZEN variable are mapped, we support three options for wild cards: *WildCardX*, *WildCardY*, and *WildCardZ* (i.e. ZEN variable is associated with the X-axis, Y-axis, respectively Z-axis). In case of a multiple metric selection, a corresponding wild card must be selected from the menu. Otherwise the last selected metric will be visualised only. Figure 7 shows a sample 4D barchart, generated by the ADV dialog-box snapshot in Fig. 6. Currently, we are in the process of integrating a 6D hierarchy-chart, which would allow the representation of two additional wild cards.

ANY matches any ZEN variable name. ANY is different from a wild card in the sense that the corresponding ZEN variable is not selected as a visualisation axis. This is useful when the value of a specific ZEN variable is uniquely determined (based on a ZEN constraint) by other ZEN variables (one of which is a wild card).

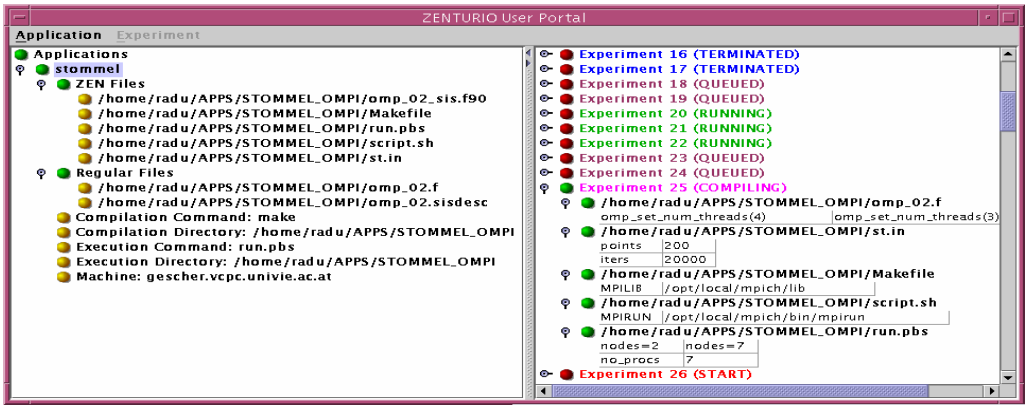


Figure 4: User Portal showing the status of a running ocean simulation application (see Section 6.1).

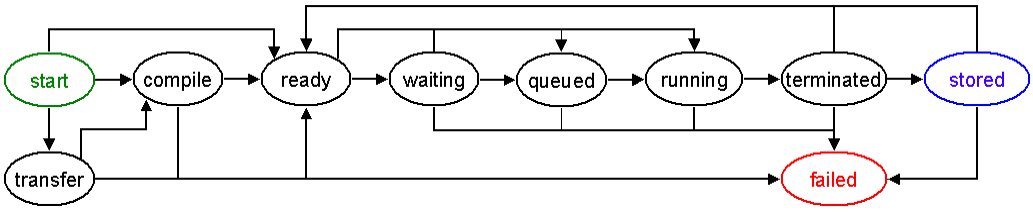


Figure 5: Experiment state transition diagram.

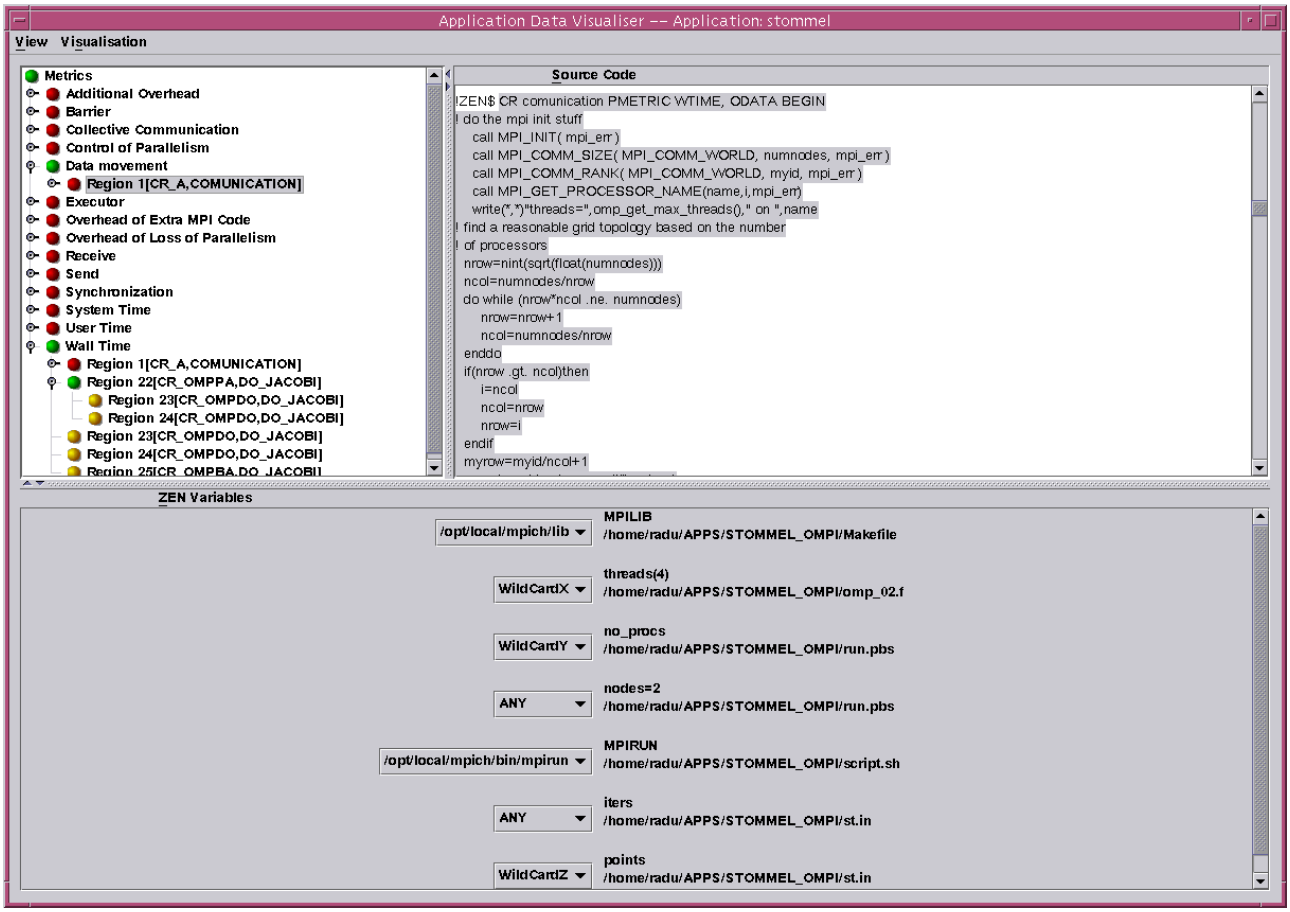


Figure 6: The Application Data Visualiser (ADV) dialog-box which generates the diagram in Fig. 7. The three wild card selections indicate the three visualisation axis: number of OpenMP threads (ZEN variable `threads(4)`), number of MPI processes (ZEN variable `no_procs`), and the problem size (ZEN variable `points`). The ZEN variables `nodes=2` and `iters` are given the `ANY` value, because their value is determined by `no_procs` and `points`, respectively. The fourth dimension is represented by the data movement (i.e. communication) metric to be visualised for the entire program. Barchart was chosen as visualisation diagram.

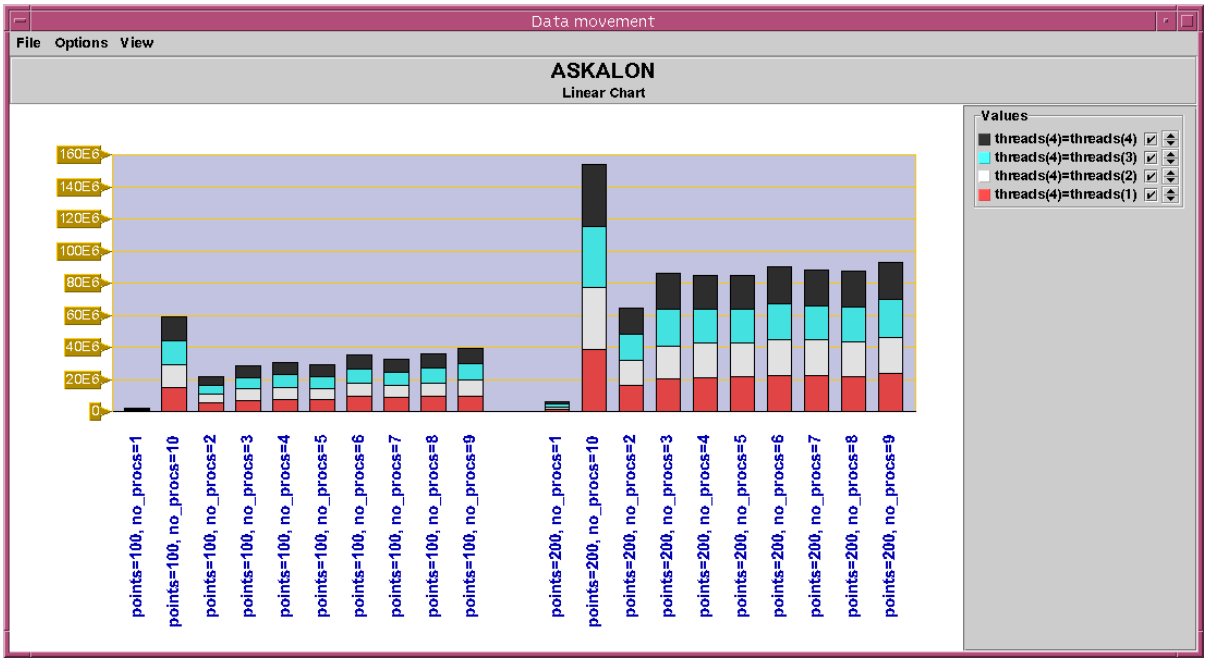


Figure 7: Sample 4D visualisation barchart for the Stommel application presented in Section 6.1. Every bar section describes the communication time for different number of threads used per SMP node (ZEN variable `thread(4)`). The two group of bars reflect the two different problem sizes analysed (ZEN variables `points` and `iters`). Each bar corresponds to the number of MPI processes used for each experiment (ZEN variables `no_procs` and `nodes=2`).

A similar dialog box is used to visualise output parameters across multiple experiments. For this purpose a program must write the output data into a file or to standard output. The user selects a file name (or standard output) which corresponds to the output parameter to be visualised. By restricting the ZEN variable values as done for the performance metric visualisation, we choose the experiment subset for which the output parameters should be visualised. Figure 10 (see Section 6.2) shows an example diagram for visualising a single output parameter.

4.3 Experiment Executor (EE)

The EE grid service has the role of executing and managing experiments on the E-site. It can operate in five different modes, depending on the scheduler with which it interacts at the back-end: (1) *fork*, for single processor machines; (2) *PBS* [27] for workstation clusters; (3) *GRAM-fork* [5] for running an experiment with the Globus [10] metacomputing infrastructure on a single Globus site, using fork as job manager; (4) *GRAM-PBS* for running an experiment with Globus on a single Globus site, using PBS as a job manager; (5) *DUROC* for running an experiment on multiple Globus sites. For clusters, the EE physically resides on the front-end node, while for Globus it can reside on any Globus site.

The EE was designed as a stand-alone Grid service independent of the ZENTURIO architecture. Thus, it can be deployed for other experiment management infrastructures as well. The EE provides functionality for: (1) adding/removing experiments; (2) compiling experiments; (3) executing experiments; (4) retrieving status information of experiments; (5) subscribing for status notification callbacks which avoids permanent polling of clients (e.g. UP); (6) terminating experiments; (7) transferring experiment output (i.e. standard output, standard error, any output file, and performance data); (8) retrieving all experiments associated with a certain application (optionally being in a certain state). These methods can be applied to individual or collective (i.e. all experiments belonging to an application) experiments.

The execution command for running an experiment must be given according to the EE's execution mode, as follows: (1) for the *fork* mode it is the command itself that runs the application; (2) for the *PBS* [27] mode it is the path to the PBS script used by the `qsub` job submission command; (3) for the *GRAM-fork*, *GRAM-PBS* and *DUROC* modes it is the path to the RSL (Resource Specification Language) [5] script used by the `globusrun` command.

Upon completion (normal or abnormal) of an experiment all of its standard output, standard error, performance data, and output files are automatically stored in the Experiment Data Repository.

Experiments can be optionally given a maximum execution time in the execution script (i.e. `walltime` option in PBS, respectively `maxCPUTime` in RSL). If not explicitly terminated, an experiment can fail either because of an abnormal termination (e.g. program or system error) or because it exceeded its maximum execution time. In order to deal with casual scheduler job processing errors or some non-deterministic application or system behaviour, the user can specify the maximum number of times EE reruns an experiment before reporting it as failed to the UP. Furthermore, the user can manually restart (failed) experiments.

5. IMPLEMENTATION ISSUES

In this section we briefly outline the implementation of the ZENTURIO architecture as described in Section 4.1. ZENTURIO is implemented as a Grid service-based architecture, which consists of a User Portal (UP) and three Grid services: the Registry Service (RS), the Experiment Generator (EG) service, and the Experiment Executor (EE) service. We are

using Sun's Jini distributed service technology [9] to build the middleware that links the Grid services with the clients in a consistent Grid environment. The Java implementation of the ZENTURIO services has been strictly separated from Jini services through Jini specific wrapper classes. The Java classes which implement the ZENTURIO services are "Jini"-ised by means of delegation (see Fig. 8). The interface to a Jini service is defined by a Java interface. Interaction between services and the UP is done based on Java remote method invocation calls. The RS is implemented by the Jini lookup service. There can be arbitrarily many lookup services installed in a network, where Jini ZENTURIO services can be registered.

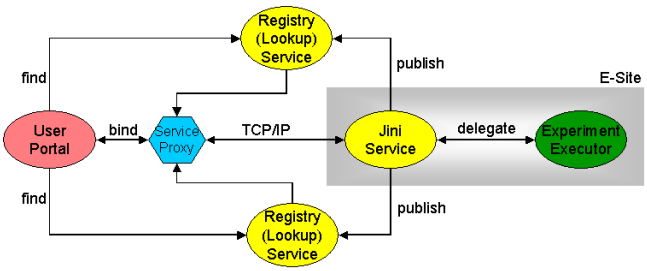


Figure 8: The ZENTURIO Jini Service Implementation.

Alternatively, ZENTURIO can also be implemented based on a Web service infrastructure. The Grid service interface is exported from Java to WSDL (Web Service Description Language) [4] by using existing translators. The ZENTURIO middleware would be based on the SOAP (Simple Object Access Protocol) [21] XML-based protocol. The RS can be implemented as a centralised component based on UDDI (Universal Description, Discovery and Integration) [26], or distributed, based on WSIL (Web Services Inspection Language) [3]. We have already deployed the ZENTURIO Grid services EE and EG – implemented in pure Java – into the WASP [23] Web Application Server. The EE and EG Grid services are available for external use at `http://gescher.vcpc.univie.ac.at:6060/admin/console` (push the `Refresh listing` button to see them). We are currently working on a more mature implementation, which tightly follows the OGSA proposal [11, 25].

6. EXPERIMENTS

In order to evaluate our prototype implementation we have conducted two different experiments: a performance study for an ocean simulation application (see Section 6.1) and a parameter study for a backward pricing application (see Section 6.2).

6.1 Experiment-1: Performance Study for an Ocean Simulation Application

In the first experiment we used an OpenMP/MPI Fortran90 version of a code that simulates the ocean in order to explain the westward intensification of wind-driven ocean currents [22]. The experiments were conducted on a cluster of SMP nodes, each node containing 4 Intel Pentium III Xeon 700 MHz CPUs. Experiments were submitted to this cluster via the Globus metacomputing environment as single jobs and used the locally installed MPICH library for communication. The jobs were scheduled on dedicated nodes through PBS [27]. We used ZEN to specify the application parameters and performance metrics of interest. Our prototype implementation automatically invoked all the corresponding experiments and stored the output results and performance data in a database. ADV (see Section 4.2) has been used to visualise performance data.

We studied the code scalability by varying the machine size, the problem size, and the interconnection network. The machine size consists of two dimensions: (1) the number of SMP nodes (1 to 10), controlled by directives inserted in the Globus RSL script (see Ex. 6.1); (2) the number of threads per SMP node (1 to 4), controlled by the input parameter to the OpenMP library routine `omp_set_num_threads` (see Ex. 6.2). The `count` parameter name in Ex. 6.1 was assigned a value equal to the number of SMP nodes times the number of processors per CPU, which was necessary to persuade GRAM to allocate the exact number of nodes needed. The shell script `script.sh` (see Ex. 6.3) sets the maximum number of MPI processes per node to 1 through the `MPI_MAX_CLUSTER_SIZE` environment variable. `MPIRUN` contains the path to the corresponding `mpirun` script. It invokes the MPI application on a number of nodes equal to the number of entries in the `$PBS_NODEFILE` machine file (i.e. the number of SMP nodes allocated by GRAM). Thus, intra-node parallelisation is handled by the OpenMP compiler. The problem size was varied by changing the grid (ocean) size and the number of iterations over the grid (see Ex. 6.4). We examined the performance of two different interconnection networks (Fast Ethernet and Myrinet) for this application. The corresponding MPI library implementation is chosen by annotating the application `Makefile`, as shown in Ex. 6.5. The path to the MPI library has been assigned to the variable `MPILIB`. The global constraint file makes the right association between the implementation specific MPI libraries and the `mpirun` scripts used (Fast Ethernet versus Myrinet, see Ex. 6.6).

For every experiment, the total execution and communication overhead has been measured (denoted by the `ODATA` and `WTIME` mnemonics in Ex. 6.2).

Example 6.1 (Globus RSL Script)

```
+(&
  (resourceManagerContact="gescher/jobmanager-pbs")
(*ZEN$ SUBSTITUTE count\=4 = { count={4:40} }*)
  (count=4)
  (jobtype=single)
  (directory="/home/radu/APPS/STOMMEL_OMPI")
  (executable="script.sh")
  (stdin="st.in")
  (stdout="st.out")
  (stderr="st.err")
 )
```

Example 6.2 (Source Code Excerpt)

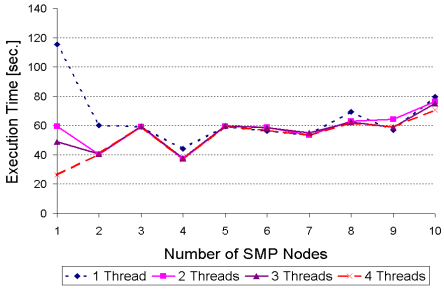
```
!ZEN$ CR CR_P PERF ODATA, WTIME
...
!ZEN$ SUBSTITUTE threads\(4\) = { threads({1:4}) }
      CALL omp_set_dynamic(.true.)
      CALL omp_set_num_threads(4)
```

Example 6.3 (script.sh)

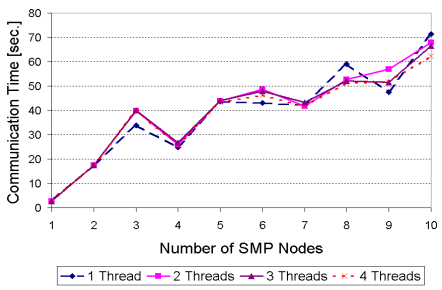
```
#!/bin/sh
setenv MPI_MAX_CLUSTER_SIZE 1
cd $PBS_O_WORKDIR
nodes = `wc -l < $PBS_NODEFILE`
#ZEN$ ASSIGN MPIRUN = { /opt/local/mpich/bin/mpirun,
                        /opt/local/mpich_gm/bin/mpirun.ch_gm }
$(MPIRUN) -np $nodes -machinefile $PBS_NODEFILE omp_02_sis
```

Example 6.4 (Input Data File – `st.in`)

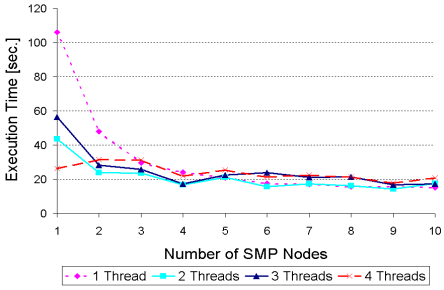
```
!ZEN$ SUBSTITUTE points = { 200, 400 }
      points points
      2000000, 40000000
      1.0e-9 2.25e-11 3.0e-6
!ZEN$ SUBSTITUTE iters = { 20000, 40000 }
      iters
!ZEN$ CONSTRAINT INDEX points == iters
```



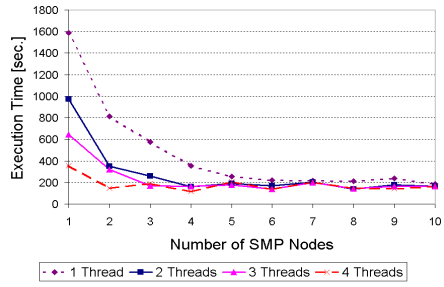
(a) 200 × 200 Grid, 20000 Iterations, Fast Ethernet



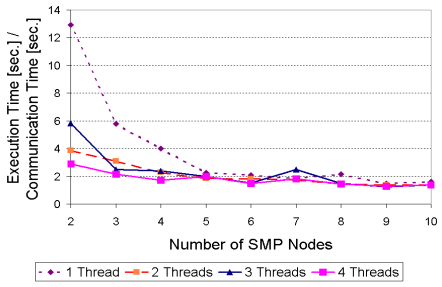
(b) 200 × 200 Grid, 20000 Iterations, Fast Ethernet



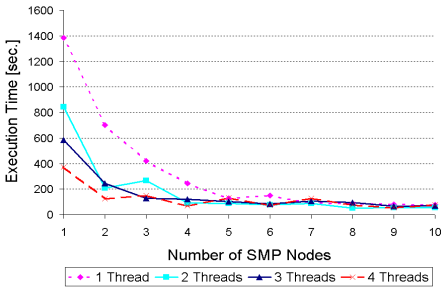
(c) 200 × 200 Grid, 20000 Iterations, Myrinet



(d) 400 × 400 Grid, 40000 Iterations, Fast Ethernet



(e) 400 × 400 Grid, 40000 Iterations, Fast Ethernet



(f) 400 × 400 Grid, 40000 Iterations, Myrinet

Figure 9: Performance study for an ocean simulation application with varying problem sizes, and machine sizes.

Example 6.5 (Makefile)

```
#ZEN$ ASSIGN MPILIB = { /opt/local/mpich/lib,
                        /opt/local/mpich_gm/lib }
...
$(TARGET): $(TARGET).o
        $(F90) $(TARGET).o -o $@ -L$(MPILIB) -lmpich
```

Example 6.6 (Global Constraint File)

```
#ZEN$ CONSTRAINT INDEX Makefile:MPILIB == script.sh:MPIRUN
```

Overall, only 11 ZEN directives had to be included in four application files and one global constraint file (see Ex. 6.6). In addition, the compilation (`make`) and execution (`run.rsl`) commands and directories, the machine name (gescher), and the scheduler (GRAM-PBS) had to be provided. A total set of 160 experiments were automatically generated and executed for several hours. The performance and the output results were automatically stored into the Experiment Data Repository. The scalability diagrams for different machine sizes, problem sizes, and interconnection networks shown in Fig. 9, were automatically generated by the ADV (see Section 4.2).

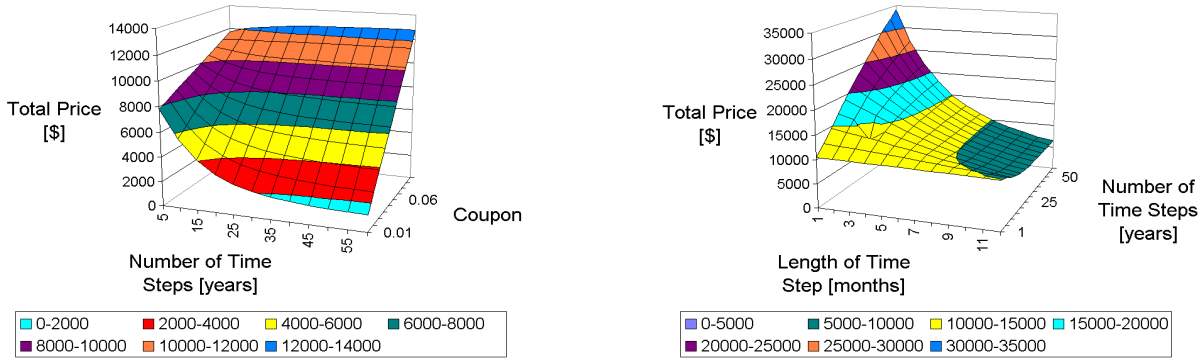
For a 200 × 200 grid, the program does not scale (see Fig. 9(a)) which is explained by the poor communication behaviour (see Fig. 9(b)) for this problem size over Fast Ethernet. We also observed that this problem size scales well with the number of threads on a single SMP node. For larger number of nodes the number of threads does not improve the overall performance. The reason is the large MPI communication overhead which dominates the overall execution time. The same problem size scales much better over Myrinet 9(c), which is due to the lower communication cost on the much faster Myrinet compared to Fast Ethernet.

The 400 × 400 grid problem size shows a very reasonable scaling behaviour over Fast Ethernet for up to 4 SMP nodes (see Fig. 9(d)). Using more than 4 SMP nodes does not substantially decrease the execution time anymore. This is due to an increased communication overhead and a decreasing ratio between execution and communication time (see Fig. 9(e)). For smaller number of nodes, the computation to communication time ratio is high and effective parallelisation of OpenMP loops onto a set of threads yields a very satisfying scaling behaviour. Increasing the number of threads decreases the execution time as expected. Similarly, the same problem size scales over Myrinet too (see Fig. 9(f)).

By using ZENTURIO the user could easily find answers to the following questions: How does the performance scale for a given application? What machine size yields the best performance on a given architecture? How many threads per node should be started? What is the best (in terms of performance) network to use for a given application?

6.2 Experiment-2: Parameter Study for the Backward Pricing Application

The backward pricing kernel [8] is a parallel implementation of the backward induction algorithm to compute 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 performed two parameter studies by varying four input parameters: (1) the coupon bond (0.01 to 0.1); (2) the number of time steps, over which the price is computed (5 to 60); (3) the coupon bond's end time, which must be equal to the number of time steps; (4) the length of one time step (1/12 to 1, with increment 1/12).



(a) Length of Time Step = 1.0

(b) Coupon = 0.05

Figure 10: Evolution of the total price for the Backward Pricing application.

Example 6.7 (Input Parameter Assignment)

```
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 }
```

The application was encoded such that all input parameters are read from different input data files. We annotated one single source file with 4 ZEN assignment directives immediately after the corresponding `read` statements (see Ex. 6.7). Thus, the original `read` statement is made redundant. An additional constraint directive is used to guarantee that the coupon bond's end time is identical with the number of time steps for every experiment. A more efficient method – similar to Ex. 6.4 – would be to annotate the input files directly instead of the source code, thus avoiding unnecessary compilation. For the sake of showing the flexibility of ZEN and ZENTURIO, we decided to show the source code instrumentation instead. Note that for this experiment we are not interested in performance aspects, but concentrate only on examining the effects of different input parameters on the corresponding output results.

A total number of 1200 experiments were generated, from which two sample diagrams are depicted in Fig. 10. The 3D surface in Fig. 10(a) shows the evolution of the total price (output parameter of the backward pricing code) as a function of the number of time steps and the coupon, with the following meaning: (a) the price decreases with the maturity (number of time steps × length of time step), because the effect of discounting future payments increases (i.e. \$100 in 20 years are less then \$100 in 10 years), but only if the coupon is less than the interest rates (e.g. for 0.06, the coupon rate is greater than the interest rates); (b) the price increases with the coupon, because the higher the coupon rate is, the higher the future payments are; (c) for very large maturities, the price linearly depends only on the coupon.

Figure 10(b) shows the price evolution by varying the number of time steps and the length of a time step, with the following explanation: (a) the price decreases with the length of a time step, because a smaller payment number implies less money in the future; (b) depending on the number of time steps, the price may increase or decrease with maturity, depending on how much the smaller number of payments are compensated by smaller discount effects.

7. CONCLUSIONS

Developing ZENTURIO has been driven by the need to support scientists and engineers in their effort to conduct parameter studies, performance analysis and tuning, and software testing for a wide variety of parallel and distributed systems, including cluster and Grid architectures. The main contributions of ZENTURIO are centered around the following aspects: (1) It employs a directive based language (ZEN) to specify arbitrary experiments by substituting values of any problem and machine parameter, to request for performance metrics for arbitrary code regions, and to control the number of experiments. ZEN directives are not changing the semantics of any file, as they are interpreted as comments and therefore ignored by systems that are not aware of them. (2) An interface to an external instrumentation engine is supported that allows to instrument applications for a wide variety of performance metrics. (3) A Grid service-based architecture has been designed and implemented based on Java/Jini and Web Service technologies. An interface to a variety of job schedulers allows us to easily add new target architectures. (4) ZENTURIO does not assume pre-compiled and pre-installed application codes on the target machine. Compilation and file transfer is supported. (5) A graphical User Portal enables the user to control and monitor experiments. A flexible technique to automatically visualise performance and output data across large sets of experiments has been developed.

Based on the experiments presented we believe that ZENTURIO has a promising potential to support performance analysis and parameter studies for cluster and Grid architectures.

Future work involves the development of a more sophisticated analysis component which compares experiments based on performance and application result data. We intend to adjust our design and our on-going Web Service based implementation towards the OGMA specification. We are also planning to extensively evaluate ZENTURIO for larger Grid architectures.

8. 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] Mark Baker and Rajkumar Buyya. Cluster computing at a glance. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 3–47. Prentice Hall PTR, Upper Saddle River, NJ, 1999. Chap. 1.

[3] Keith Ballinger, Peter Brittenham, Ashok Malhotra, William A. Nagy, and Stefan Pharies. Web Services Inspection Language (WS-Inspection) 1.0 . `http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html`.

[4] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL). Technical report, The World Wide Web Consortium, March 2001. `http://www.w3.org/TR/wsdl`.

[5] 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.

[6] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January/March 1998.

[7] Thierry Delot, Pascal Déchamboux, Béatrice Finance, Yann Lepetit, and Gilles LeBrun. LDAP, databases and distributed objects: Towards a better integration. *Lecture Notes in Computer Science*, 2209:140–148, 2001.

[8] 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.

[9] W. K. Edwards. Core Jini. *IEEE Micro*, 19(5):10–10, September/October 1999.

[10] 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.

[11] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, The Globus Project and The Global Grid Forum, January 2002. `http://www.globus.org/research/papers/OGSA.pdf`.

[12] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces principles, patterns, and practice.* Addison-Wesley, Reading, MA, USA, 1999.

[13] High Performance Fortran Forum, High Performance Fortran Language Specification. Version 2.0.δ, Technical Report, Rice University, Houston, TX, Jannuary 1997.

[14] I. Foster and C. Kesselman and S. Tuecke. The Anatomy of the Grid. *The International Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[15] 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.

[16] 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.

[17] Heather Kreger. Web Services Conceptual Architecture (WSCA 1.0). Prepared for Sun Microsystems, Inc., IBM Software Group, May 2001.

[18] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, January 1996.

[19] Radu Prodan and Thomas Fahringer. ZEN: A Directive-based Language for Automatic Experiment Management of Parallel and Distributed Programs. In *Proceedings of the 2002 International Conference on Parallel Processing (ICPP-02), Vancouver, Canada*. IEEE Computer Society Press, August 2002. To appear.

[20] M. Romberg. The UNICORE architecture: Seamless access to distributed resources. *Proceedings of the 8th International Symposium on High Performance Distributed Computing HPDC-8*, pages 287–293, August 1999.

[21] 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.

[22] H.M. Stommel. The western intensification of wind-driven ocean currents. *Transactions American Geophysical Union*, 29:202–206, 1948.

[23] Systinet. Web Applications and Services Platform. `http://www.systinet.com/products/`.

[24] 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 *Proceeding SC'2001, Denver, USA*, November 2001.

[25] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. Grid service specification. Technical report, The Globus Project and The Global Grid Forum, February 2002. `http://www.globus.org/research/papers/gsspec.pdf`.

[26] UDDI: Universal Description, Discovery and Integration. `http://www.uddi.org`.

[27] Veridian Systems. PBS: The Portable Batch System. `http://www.openpbs.org`.

[28] 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.