ZEN: A Directive-based Language for Automatic Experiment Management of Distributed and Parallel Programs*

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

Performance-oriented code development, software testing, performance analysis and parameter studies for distributed and parallel systems commonly require to conduct a large number of executions. Every execution of an application can be viewed as a scientific experiment. So far there exists very little support to specify and to control execution of a large number of experiments. Various problems must be addressed, such as which input files to read, where to store program's output, what performance metrics to measure and what range of problem parameters to observe. This paper describes ZEN, a directive-based language to support automatic experiment management for a wide variety of parallel and distributed architectures. It is used to specify arbitrarily complex program executions in the context of performance analysis and tuning, parameter studies, and software testing. ZEN introduces directives to substitute strings and insert assignment statements inside arbitrary files, such as program, input, script, or makefiles. This 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. The number of experiments can be controlled through ZEN constraint directives. Finally, the programmer may request a large set of performance metrics to be computed for any code region of interest. The scope of ZEN directives can be restricted to arbitrary file or code regions.

We have implemented a prototype tool that supports the user by employing ZEN directives to control and manage parameter studies and performance experiments for distributed and parallel programs on cluster architectures. We report results of using our prototype implementation for performance analysis of an ocean simulation application and for parameter study of a computational finance code.

Keywords: directive-based language, experiment management, parameter study, performance analysis, parallel and distributed computing.

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

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 [7], is a tedious, time-consuming and error-prone undertaking.

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, researchers are forced to 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 and code regions, etc.

In this paper we describe ZEN, a directive-based language for automatic experiment management. Its goal is to support users in specifying arbitrary complex program executions, and to control and guide parameter studies, performance analysis, and software testing on a wide variety of parallel and distributed architectures. ZEN directives are comment-based and ignored by systems that are unaware of their semantics. ZEN directives may substitute strings, insert assignment statements in arbitrary files (such as program, input, script, scheduling, or makefiles), and control the number of experiments. This enables the programmer to specify value ranges for problem, system or machine parameters. Such parameters include program variables, file names, compiler options, target machines, machine sizes, scheduling strategies, data distributions, etc. Performance behaviour directives can be used to gather performance metrics. The scope of ZEN directives can be restricted to arbitrary file or code regions.

We have implemented a prototype that parses application files annotated with ZEN directives and generates appropriate application codes, based on the semantics of the directives. The applications are then transferred to the target machine for compilation and execution. Upon their completion, the output files and performance data are stored into a database for post-mortem analysis and visualisation. Apart from annotating files with ZEN directives and providing information about the compilation and execution commands, all of the above steps are processed fully automatically.

Currently, we are in the process of evaluating ZEN and our experiment management prototype system for several scientific and engineering applications on cluster architectures.

The rest of this paper is organised as follows. The next section discusses related work. Section 3 presents the ZEN language in detail. Section 4 briefly delineates the design of our prototype experiment management system. Experiments are presented in Section 5. Finally, some concluding remarks are made and future work is outlined in Section 6.

2 Related Work

Our work on a language specification for automatic experiment management is centered around three different research areas: scientific experiment management, performance analysis, and parameter studies.

The Paradyn performance analysis tools [9] supports experiment management through: (1) a representation for the execution space of performance experiments; (2) techniques for quantitatively comparing several experiments; and (3) performance diagnosis based on dynamic instrumentation. Experiments (by varying problem and machine size parameters) have to be set up manually, whereas performance analysis is done automatically for every different experiment. Paradyn's analysis is based on function calls only, whereas ZEN supports performance analysis for generic code regions.

The ZOO project [8] 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. A transformation mechanism maps the contents of a database to application specific input files and vice versa for output files. ZEN is not restricted to input files but enables also the parametrisation of arbitrary strings in any file, including source files, makefiles, etc.

Nimrod [1] is a tool that manages the execution of parameter studies. A parameterised experiment is specified by a plan file. Nimrod generates one job for each unique combination of parameter values, by taking the cartesian product of all values. Unlike in ZEN, the set of possible parameter value combinations cannot be restricted. Moreover, Nimrod does not allow to substitute strings in program files.

Instead of using a language based approach, the ILAB [18] project tries to control parameter studies through a graphical user interface. ILAB is restricted to input files.

The Unicore project [13]) facilitates the usage of supercomputers on the Internet. The user manually controls remote compilation or transfers binaries, specifies command-line parameters, and indicates locations for input/output files. However, there is no support for automatic invocation of multiple experiments, as targeted by ZEN.

SKaMPI [12] provides a benchmarking environment for MPI codes, with the goal of analysing the runtime of MPI operations. A run-time parameter file allows the description of a suite of measurements based on specific parameter values. Only a fixed number of machine and problem size parameters can be controlled by the programmer. SKaMPI provides limited support for performance analysis.

3 ZEN Language Description

ZEN is a directive-based language for the specification of arbitrary complex program executions to support parameter studies, performance analysis and tuning, software testing, etc. A directive-based 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, the scope of directives can be global or limited to specific code regions. Directives are comment-based and thus do not change the semantics of the code, as they are ignored by systems that are unaware of their semantics. ZEN directives are language independent. Only the ZEN transformation system (ZTS – see Section 4), which generates source code for experiments based on ZEN directives, must be aware of the programming language of ZEN instrumented program files. ZEN allows to express a larger number of value ranges for arbitrary parameters or program variables in a very compact form, for instance by using triplet notation (see Section 3.1). This feature enables the generation of large number of experiments with very few short directives. Constraint directives are introduced to control the number of experiments in order to avoid experiments that do not produce any useful results. In the remainder of this section, we give an introduction to ZEN sets and directives, before discussing all individual ZEN directives in detail.

3.1 ZEN Sets

ZEN enables the specification of value ranges for arbitrary parameters, variables, or strings in a file, before experiments are invoked. ZEN introduces the so-called ZEN sets to efficiently describe these value ranges. A *ZEN set* is a totally ordered set of (integer or real) numbers or strings, with a well-defined syntax and a well-defined evaluation function ε. The following syntax based on regular expressions is used to specify ZEN sets:

zen-set **is** “{” *elem-list* “}”

elem **is** *num*

or *comp-elem*

num **is** *low*:*up*[:*stride*]

or *number*

comp-elem **is** (*zen-num-set* | *zen-string*)+

low **is** *number*

up **is** *number*

stride **is** *number*

number **is** *integer*

or *real*

zen-num-set **is** “{” *num-list* “}”

zen-string **is** ([^\n{},:] | “\{” | “\}” | “\,” | “\:”)*

A term *elem-list* corresponds to a list of *elem* terms, separated by commas (the same holds for *num-list*).

Let $\cdot$ denote the string concatenation operator, also referred in the following using one blank character, and let $\mathcal{P}$ denote the power set and $\mathbb{R}$ the set of real numbers. The semantics (i.e. the concrete set of elements) of a ZEN set, is given by the evaluation function: $\varepsilon : \textit{zen-set} \rightarrow \mathcal{P}(\mathbb{R} \cup \textit{string}),\ \varepsilon(\{elem_1, \ldots, elem_n\}) = \bigcup_{1 \leq i \leq n} \overline{\varepsilon}(elem_i)$, where function $\overline{\varepsilon}$ is defined in Figure 1. The pattern *low:up:stride* has by default $stride = 1$, therefore $\overline{\varepsilon}(\textit{low:up}) = \overline{\varepsilon}(\textit{low:up:1})$. *zen-strings* must obey the syntax defined by the evaluation function $\overline{\varepsilon}_s$. Commas, braces and colons inside *zen-strings* must be prefixed with one '\' character. This distinguishes commas from the value delimiters of a *zen-num-set*, avoids *zen-num-set*s inside *zen-string*s and allows the pattern *low:up:stride* be a *zen-string*. Figure 2 contains a few ZEN set evaluation examples.

$$\overline{\varepsilon} : \textit{elem} \rightarrow \mathcal{P}(\mathbb{R} \cup \textit{string}),\ \overline{\varepsilon}(e) = \begin{cases} \left\{\textit{low} + k * \textit{stride} \mid k \in \mathbb{N}, k \in \left[0 \,..\, \frac{\textit{up}-\textit{low}}{\textit{stride}}\right]\right\}, & \textit{e is low:up:stride}; \\ \{e\}, & \textit{e is number}; \\ X, & \textit{e is } (\textit{zen-num-set} \mid \textit{zen-string})+, \end{cases}$$

$$(\textit{zen-num-set} \mid \textit{zen-string})+ \ = \ \textit{zen-string}_1\ \{\textit{num}_{11}, \ldots, \textit{num}_{1n_1}\} \ldots \textit{zen-string}_p\ \{\textit{num}_{p1}, \ldots, \textit{num}_{pn_p}\}\ \textit{zen-string}_{p+1},$$

$$X = \{\overline{\varepsilon}_s(\textit{zen-string}_1)\ n_1 \ldots \overline{\varepsilon}_s(\textit{zen-string}_p)\ n_p\ \overline{\varepsilon}_s(\textit{zen-string}_{p+1}) \mid$$

$$\forall\ (n_1, \ldots, n_p) \in \varepsilon(\{\textit{num}_{11}, \ldots, \textit{num}_{1n_1}\}) \times \ldots \times \varepsilon(\{\textit{num}_{p1}, \ldots, \textit{num}_{pn_p}\})\},$$

$$\overline{\varepsilon}_s : \textit{string} \rightarrow \textit{string},\ \overline{\varepsilon}_s(s) = \begin{cases} s, & \forall\, e \in \{\text{``}\backslash,\text{``}, \text{``}\backslash\{\text{``}, \text{``}\backslash\}\text{``}, \text{``}\backslash:\text{``}\},\ e \notin s; \\ \overline{\varepsilon}_s(s_l)\ c\ \overline{\varepsilon}_s(s_r), & s = s_l\ \backslash c\ s_r,\ \forall\, c \in \{\text{`},\text{`}, \text{`}\{\text{`}, \text{`}\}\text{`}, \text{`}:\text{`}, \text{``}\}; \end{cases}$$

Figure 1. The ZEN set element evaluation function.

$$
\begin{aligned}
\varepsilon(\{1,2,3\}) &= \{1,2,3\};\\
\varepsilon(\{1:10:2\}) &= \{1,3,5,7,9\};\\
\varepsilon(\{1\backslash:10\backslash:2\}) &= \{1:10:2\};\\
\varepsilon(\{0,1:10:2,11\}) &= \{0,1,3,5,7,9,11\};\\
\varepsilon(\{foo(\{10,20,30\})\}) &= \{foo(10), foo(20), foo(30)\};\\
\varepsilon(\{foo(\backslash\{10\backslash,20\backslash,30\backslash\})\}) &= \{foo(\{10,20,30\})\};\\
\varepsilon(\{\texttt{BLOCK}(\{4:10:2\}), \texttt{CYCLIC}(\{8,16\})\}) &= \{\texttt{BLOCK}(4), \texttt{BLOCK}(6), \texttt{BLOCK}(8), \texttt{BLOCK}(10), \texttt{CYCLIC}(8), \texttt{CYCLIC}(16)\};\\
\varepsilon(\backslash\{\texttt{BLOCK}(\{4:10:2\}), \texttt{CYCLIC}(\{8,16\})\backslash\}) &= \{\{\texttt{BLOCK}(\{4:10:2\}), \texttt{CYCLIC}(\{8,16\})\}\};\\
\varepsilon(\{\texttt{BLOCK}(\backslash\{4:10:2\backslash\}), \texttt{CYCLIC}(\backslash\{8,16\backslash\})\}) &= \{\texttt{BLOCK}(\{4:10:2\}), \texttt{CYCLIC}(\{8,16\})\};\\
\varepsilon(\{A(\{0:10:5\}\backslash, 4\backslash:12\backslash:4)\}) &= \{A(0,4:12:4), A(5,4:12:4), A(10,4:12:4)\};\\
\varepsilon(\{A(\{0:10:5\}\backslash, \{4:12:4\})\}) &= \{A(0,4), A(0,8), A(0,12), A(5,4), A(5,8), A(5,12), A(10,4), A(10,8), A(10,12)\}.\\
\varepsilon(\{\texttt{STATIC}\backslash, \{4,8\}, \texttt{DYNAMIC}\backslash, \{1:4\}\}) &= \{\textit{STATIC,4}\ , \textit{STATIC,8}\ , \textit{DYNAMIC,1}\ , \textit{DYNAMIC,2}\ , \textit{DYNAMIC,3}\ , \textit{DYNAMIC,4}\}.
\end{aligned}
$$

Figure 2. ZEN set evaluation examples.

- $\forall\, elem_i, elem_j \in \varepsilon(\{elem_1, \ldots, elem_n\}),\ i,j \in [1\,..\,n],\ elem_i \prec elem_j \iff i < j;$
- $\forall\, e_i, e_j \in \overline{\varepsilon}(\textit{low:up:stride}), e_i \prec e_j \iff e_i = low + k_i * stride \ \wedge\ e_j = low + k_j * stride \ \wedge\ k_i < k_j;$
- $\forall\, (n_1, \ldots, n_p), (n'_1, \ldots, n'_p) \in \varepsilon(\{num_{11}, \ldots, num_{1n_1}\}) \times \ldots \times \varepsilon(\{num_{p1}, \ldots num_{pn_p}\}),$

  $string_1\ n_1 \ldots string_p\ n_p\ string_{p+1} \prec string_1\ n'_1 \ldots string_p\ n'_p\ string_{p+1} \iff$

  $\exists\, i \in [1\,..\,n]$ *such that* $(\forall\, j \in [1\,..\,i-1] : n_j = n'_j) \ \wedge\ n_i \prec n'_i;$
- $\forall\, (A, \prec),\ (B, \prec)$ *totally ordered sets* $\implies (A \cup B, \prec)$ *totally ordered set, where:*

  $\forall\, a, b \in A \cup B,\ a \prec b \iff a \in A \ \wedge\ b \in B \backslash A \ \vee\ (a, b \in A \ \vee\ a, b \in B \backslash A) \ \wedge\ a \prec b.$

Figure 3. Rules for expressing the total order of elements in ZEN sets.

A totally ordered set $\mathcal{S}$ with an ordering operation $\prec$ is denoted by $(\mathcal{S}, \prec)$. A single element of a ZEN set is called *ZEN element*. The *total order* of ZEN elements in a ZEN set, denoted by the operator $\prec$, is given by the ordering rules expressed in Figure 3. The total order of ZEN sets is used by the ZEN index constraint directive (see Section 3.7).

3.2 ZEN Directives

A *ZEN directive* is a comment line starting with the prefix `ZEN$`. The characters which mark the begin (and eventually the end) of a comment is the only programming language specific feature of ZEN. In order to define a ZEN directive, a comment line must have the `ZEN$` prefix immediately after the special character(s) which mark its begin. Example 3.1 shows six sample ZEN directives, valid, in descending order in the context of the following programming languages: `Fortran90`, `Fortran77`, `C++` (or `Java`), `C`, `Lisp` and `shell` scripting language.

Example 3.1 (Sample ZEN Directives)

```
!ZEN$  A = { 1, 2, 3 }
CZEN$  A = { 1, 2, 3 }
//ZEN$ A = { 1, 2, 3 }
/*ZEN$ A = { 1, 2, 3 }*/
;ZEN$  A = { 1, 2, 3 }
#ZEN$  A = { 1, 2, 3 }
```

There are four categories of ZEN directives. *Substitute* (see Section 3.5) and *assignment directives* (see Section 3.3) assign a ZEN set to a so called *ZEN variable* (defined as a sequence of characters). Each ZEN element in the ZEN set represents an experimental value for the corresponding ZEN variable. The *constraint directives* (see Section 3.7) define boolean conditions over ZEN variables, thus restricting the set of possible experiments. The *performance measurement directives* (see Section 3.8) are used to request for a large number of performance metrics for specific code regions.

Every ZEN directive d, except the assignment directive, is associated with a scope denoted by *scope(d)*, which refers to the code region in which the directive holds or is applied.

The ZEN variable name must obey the following syntax constraints: (1) equality characters must be prefixed with a '\' character, which distinguishes them from the value assignment characters inside a ZEN directive (e.g. `PARAMETER::\=0`, see Section 3.5); (2) arithmetical (+, $-$, *, /, %, ^), relational (==, !=, $<$, $>$, $<=$, $>=$) and logical(!, &&,||) operators, as well as left and right parenthesis must be prefixed with a '\' character, which distinguishes them from the parenthesis and operators in a ZEN constraint (e.g. `BLOCK\(4\)`, see Section 3.7); (3) it must not have the syntax of an integer or real number, which avoids ambiguities inside ZEN constraint boolean expressions (e.g. -10.00, see Section 3.7).

ZEN variables can be of three different types: *integer*, *real* and *string*. This is determined by ZTS (see Figure 4) in the parsing phase, based on the values of the associated ZEN elements. The introduction of the integer and real types along side string (which otherwise may suffice) is motivated by need to apply value set constraints over the ZEN element values (see Section 3.7).

We present now several definitions which are important for understanding the semantics of ZEN directives.

Definition 3.1 The *value set* of a ZEN variable z, denoted by $\mathcal{V}^z$, is the totally ordered ZEN set $(\mathcal{S}, \prec)$ associated with z: $\mathcal{V}^z = \varepsilon(\mathcal{S})$ (see Section 3.1), where $\prec$ is defined in Figure 3.

Definition 3.2 The *index domain* of a ZEN variable z, denoted by $\mathcal{I}^z$, is the totally ordered set of elements $\mathcal{I}^z = (\mathcal{S}, <)$, where $\mathcal{S} = \{i \in \mathbb{N}^* \mid i \leq |\mathcal{V}^z|\}$ and $\mathbb{N}$ denotes the natural numbers. The total order of elements in $\mathcal{I}^z$ is the natural element order.

Definition 3.3 The *value function* of a ZEN variable z is the total bijective function: $\vartheta : \mathcal{I}^z \to \mathcal{V}^z$, which associates each element $i \in \mathcal{I}^z$ with the ith element $e \in \mathcal{V}^z$, such that: $\forall\, i, i_1, i_2 \in \mathcal{I}^z,\ i_1 < i < i_2 \iff \vartheta(i_1) \prec \vartheta(i) \prec \vartheta(i_2)$. We denote i to be the *index* of $e \in \mathcal{V}^z$. The *index function* $\vartheta^{-1} : \mathcal{V}^z \to \mathcal{I}^z$ is the inverse of the value function.

The value and index functions are used by the index domain constraints in Section 3.7.

An arbitrary file $\mathcal{F}$ (e.g. source, input data and makefile) augmented with a set of ZEN directives is called *ZEN file*. The ZEN file is denoted by $\mathcal{F}(z_1, \ldots, z_n)$, where z_i, $i \in [1 \mathinner{..} n]$, are the ZEN variables corresponding to the ZEN directives in $\mathcal{F}$. A *ZEN file instance*, denoted by $\mathcal{F}'(e_1, \ldots, e_n)$, where $e_i \in \mathcal{V}^{z_i}$, $i \in [1 \mathinner{..} n]$, is an instantiation of the ZEN file $\mathcal{F}$, obtained by instantiating each ZEN variable with one ZEN element from its value set.

The translation of a ZEN file to a ZEN file instance is done by the *ZEN Transformation System (ZTS)*, depicted in Figure 4. ZTS can be considered a source-to-source compiler. The scanner and parser modules examine ZEN directives, based on which an intermediate file representation (i.e. a simplified abstract syntax tree) is constructed. The ZEN unparser is different from a conventional code generator, as it commonly generates a possibly large number of file instances. The code generation rules are specified based on the semantics of the ZEN assignment and substitution directives, presented in Sections 3.3 and 3.5. The number of the ZEN file instances is given by the cardinality of the value set of a ZEN file, as defined in Section 3.6.

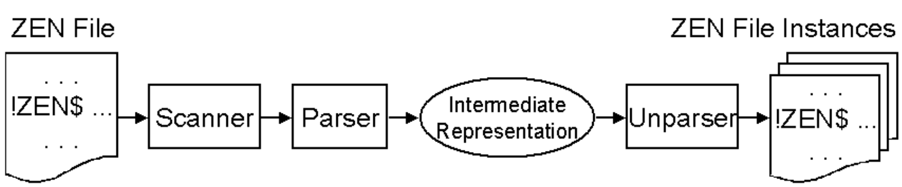


Figure 4. The ZEN Transformation System (ZTS).

3.3 ZEN Substitute Directive

The *ZEN substitute directive* is used to replace strings in ZEN files. This is expressed by assigning a ZEN set to a ZEN variable. This directive is commonly employed to examine different problem and machine sizes, data or work distributions, scheduling strategies, etc. The scope of the *global substitute directive* comprises the entire ZEN file in which it occurs, with the following syntax:

substitute-directive **is** `SUBSTITUTE` *zen-var* = *zen-set*

ZTS replaces all occurrences in the entire file of the name $\nu(z)$ of a ZEN substitute variable z with one element $e \in \mathcal{V}^z$. It is the user's task to ensure that the substitution does not change the semantics of the code.

Example 3.2 (OpenMP Loop Distribution)

```
!ZEN$ SUBSTITUTE STATIC = { STATIC\,{1,10:100:10}, DYNAMIC\,{1,10:100:10},
                            GUIDED\,{1,10:100:10} }
!$OMP DO SCHEDULE(STATIC)
...
!$OMP END DO
```

OpenMP [3] is a directive based language which represents the de-facto standard for programming of shared memory architectures. The ZEN directive in Example 3.2 allows the programmer to examine various loop scheduling strategies combined with chunk sizes. The original OpenMP scheduling clause `STATIC` is replaced with every ZEN element in the set $\mathcal{V}^{\texttt{STATIC}}$ with cardinality 33. `STATIC` means that the iterations are assigned to threads statically, before getting executed; `DYNAMIC` means that as each thread finishes a set of iteration space, it dynamically gets the next one; `GUIDED` means that the iteration space is divided into pieces such that the size of each successive piece is exponentially decreasing [3]. Note that the code shown in this example is semantically valid for both ZEN aware (i.e. understands ZEN syntax and semantics) systems and ZEN unaware (i.e. ignores all ZEN directives) compilers. The ZEN file instances generated are as depicted in Figure 5.

File Instance ($\mathcal{F}'$)	**Generated Code**
$\mathcal{F}'$(`STATIC`, 1)	`!$OMP DO SCHEDULE(STATIC,1)`
$\mathcal{F}'$(`STATIC`, 100)	`!$OMP DO SCHEDULE(STATIC,100)`
...	...
$\mathcal{F}'$(`GUIDED`, 1)	`!$OMP DO SCHEDULE(GUIDED,1)`
$\mathcal{F}'$(`GUIDED`, 100)	`!$OMP DO SCHEDULE(GUIDED,100)`

Figure 5. The file instances generated by Example 3.2.

Similarly, one could vary other OpenMP parameters, like the number of threads within a parallel region.

3.4 Local Substitute Directive

The local ZEN substitute directive restricts the scope of the global version to a subset of the ZEN file. It has the following syntax:

local-subst-dir **is** `SUBSTITUTE` *zen-var* = *zen-set* `BEGIN`
code-region
`END SUBSTITUTE`

The difference between this directive and the global version is that it is restricted to the enclosed code region, which can be any block of code. There can be nested local substitute directives as well.

HPF [6] is a directive based language for data parallel programming. The core language constructs deal with the distribution of data onto a set of parallel processors.

Example 3.3 (HPF Array Distribution)

```
!ZEN$ SUBSTITUTE BLOCK\(4\) = { BLOCK({4:10:2}), CYCLIC({10,20}) } BEGIN
!HPF$ DISTRIBUTE A(BLOCK(4)) ONTO procs
!ZEN$ END SUBSTITUTE
```

The ZEN directive in Example 3.3 defines a ZEN variable called `BLOCK(4)`, with the value set: $\mathcal{V}^{\texttt{BLOCK(4)}} = \{$ `BLOCK(4)`, `BLOCK(6)`, `BLOCK(8)`, `BLOCK(10)`, `CYCLIC(10)`, `CYCLIC(20)` $\}$. Every ZEN element represents a specific HPF distribution pattern that substitutes the original `BLOCK(4)` array distribution.

ZEN substitute directives can be used to examine many different options for other HPF directives (e.g. `PROCESSORS`, `ON HOME`, `REDISTRIBUTE`) as well.

If the textual names of two or more ZEN variables in a ZEN file is identical, these ZEN variables are called *homonym*.

Example 3.4 (Homonym ZEN Variables)

```
d1:     !ZEN$ SUBSTITUTE P\%size = { 100:1000:100 }
        ...
d2:     !ZEN$ SUBSTITUTE P\%size = { 2**{6:10} }
        ...
        !ZEN$ END SUBSTITUTE
```

The code fragment in Example 3.4 contains two ZEN directives *d1* and *d2*, which assign different ZEN sets to two ZEN variables with the identical name *P%size* (the use of '\' to prefix the '%' character was explained in Section 3.2). Note that *P%size* denotes a field in a Fortran90 [10] compound structure. The ZEN set of the second directive uses Fortran90 syntax to express powers of 2: $\{2{*}{*}i \mid i \in [6 \mathinner{..} 10]\}$. Despite the fact that the two occurrences of *P%size* semantically refer to the same program variable with a unique memory location, they effectively represent two different ZEN variables. Each has its own value set.

The impact of the homonym ZEN variables to the semantics of global and local ZEN substitution directives is as follows. No homonym global ZEN substitute variables are allowed within one ZEN file. A local substitute directive d_i with a substitution variable z_i overrides any global or local ZEN substitute directive d_j with an associated substitution variable z_j, where z_i and z_j are homonym. In this case the value set of z_i is augmented with the value set of z_j. The union of two ordered value sets is defined in Figure 3.

A ZEN variable z is therefore characterised by the: (1) textual *name*, denoted in the following by $\nu(z)$; (2) associated *ZEN directive* d which assigns a value set to z; (3) *ZEN file* $\mathcal{F}$ which contains the directive d; and (4) *line number* of the directive in $\mathcal{F}$.

In the remainder of this paper, these are naming conventions for ZEN variables: (1) if no homonym ZEN variable has been defined, we use the textual name of the ZEN variable; (2) if other homonym ZEN variables have been defined, we use the ZEN variable name subscripted with a ZEN directive identifier.

The substitution directive must be used with care, as it might replace undesired occurences of the ZEN variable in the corresponding scope (see Example 3.5). For instance, if a variable D must be substituted in a given scope, then every occurence of this character would be replaced, even in keywords such as `DO` or `END`. This problem is in particular critical for short variable names. In order to overcome this limitation, we introduced the ZEN assignment directive, described in the next section.

3.5 ZEN Assignment Directive

The *ZEN assignment directive* is used to insert assignment statements into files. Its function is to indicate all values of interest for specific problem or machine size parameters (variables) occurring in a program.

Formally, a ZEN assignment directive assigns a ZEN set to a ZEN variable which must be a valid program variable in the context of a ZEN file. The ZEN assignment directive has the following syntax:

assign-directive	**is**	`ASSIGN` *zen-var* = *zen-set*
zen-var	**is**	*string*

The ZTS textually replaces a ZEN assignment directive with an assignment statement, which assigns one element $e \in \mathcal{V}^z$ to the ZEN variable z. The assignment statement inserted by ZTS must conform to the programming language of the associated ZEN file. For example, if the ZEN file represents a C program, the assignment statement adheres to the C syntax. ZTS does not apply any type checking, or examine whether the (ZEN) variable has been declared in the program. A possible syntax error will be detected by a subsequent compilation process of the ZEN file instance.

Example 3.5 (ZEN Variable Assignment)

```
        INTEGER N, i
s1:     D = 50
        !ZEN$ ASSIGN D = { 10, 20, 30, 40, 50:100:10 }
        DO i = 1, D
        ...
        END DO
```

Example 3.5 contains one ZEN directive, which assigns 10 values to a ZEN variable D, representing the upper bounds of the immediately following `DO` loop. The value set for the ZEN variable D is: $\mathcal{V}^D = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$, the index domain: $\mathcal{I}^D = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, and the value function: $\vartheta : \mathcal{I}^D \rightarrow \mathcal{V}^D$, $\vartheta(i) = 10\,i$. Note that the code is semantically valid for both ZEN aware and ZEN unaware systems. ZEN aware systems replace the ZEN directive with an assignment of the (ZEN) variable D with one of the elements in $\mathcal{V}^D$. In this case, the assignment *s1* in the previous example becomes redundant and is subject to dead-code elimination. Also note that using a substitution instead of the assignment directive would also replace the character `D` in the keyword `DO`. The consequence would be an erroneous program.

3.6 Multi-dimensional Value Set

So far we have mostly described ZEN files with a single ZEN directive that generates a number of ZEN file instances equal to the cardinality of its value set. The ZEN file instances refer to the set of experiments implied by the ZEN directives in the ZEN file. In this section we describe how to deal with multiple ZEN directives and how to define the corresponding ZEN file instances that must be generated to conduct all experiments.

Definition 3.4 The *multi-dimensional value set* of *n* ZEN variables $z_1, \ldots, z_n$, denoted by $\mathcal{V}(z_1, \ldots, z_n)$, is the cartesian product of their value sets: $\mathcal{V}(z_1, \ldots, z_n) = \mathcal{V}^{z_1} \times \ldots \times \mathcal{V}^{z_n}$.

The *value set* of a ZEN file $\mathcal{F}(z_1, \ldots, z_n)$, denoted in the following by $\mathcal{V}(\mathcal{F}(z_1, \ldots, z_n))$, or simply by $\mathcal{V}^{\mathcal{F}}$, is the set of all ZEN file instances generated from the multi-dimensional value set of its ZEN variables: $\mathcal{V}(\mathcal{F}(z_1, \ldots, z_n)) = \{\mathcal{F}'(e_1, \ldots, e_n) \mid (e_1, \ldots, e_n) \in \mathcal{V}(z_1, \ldots, z_n)\}$.

Example 3.4 defines two ZEN directives *d1* and *d2* which, respectively, assign two homonym ZEN variables $P\%size_{d1}$ and $P\%size_{d2}$. The multi-dimensional value set for the two variables is given by the cartesian product of their value sets: $\mathcal{V}(P\%size_{d1}, P\%size_{d2}) = \mathcal{V}^{P\%size_{d1}} \times \mathcal{V}^{P\%size_{d2}}$, with the cardinality $|\mathcal{V}(P\%size_{d1}, P\%size_{d2})| = 500$.

A *ZEN application*, denoted by $\mathcal{A}(\mathcal{F}_1, \ldots, \mathcal{F}_n)$, or simply by $\mathcal{A}$, is a set of ZEN files $\mathcal{F}_1, \ldots, \mathcal{F}_n$. A *ZEN application* is commonly a full application with a set of source files, annotated with ZEN directives. A *ZEN application instance*, denoted by $\mathcal{A}'(\mathcal{F}'_1, \ldots, \mathcal{F}'_n)$, or simply by $\mathcal{A}'$, is a set of ZEN file instances which instantiate each ZEN file of the ZEN application: $\mathcal{A}'(\mathcal{F}'_1, \ldots, \mathcal{F}'_n) = \{\mathcal{F}'_i \in \mathcal{V}^{\mathcal{F}_i} \mid i \in \mathbb{N},\ i \in [1\,..\,n]\}$.

The *value set* of a ZEN application, denoted in the following by $\mathcal{V}(\mathcal{A}(\mathcal{F}_1, \ldots, \mathcal{F}_n))$ or simply by $\mathcal{V}^{\mathcal{A}}$, is the set of application instances generated by the cartesian product of the value sets of its constituent ZEN files: $\mathcal{V}(\mathcal{A}(\mathcal{F}_1, \ldots, \mathcal{F}_n)) = \{\mathcal{A}'(\mathcal{F}'_1, \ldots, \mathcal{F}'_n) \mid (\mathcal{F}'_1, \ldots, \mathcal{F}'_n) \in \mathcal{V}^{\mathcal{F}_1} \times \ldots \times \mathcal{V}^{\mathcal{F}_n}\}$.

The value set of a ZEN application represents the complete set of ZEN application instances. An application instance corresponds to a set of files that are compiled by a target compiler and executed on a target machine. The execution of a specific application instance, during which performance and output data are collected, is called an *experiment*.

3.7 ZEN Constraint Directive

Multi-dimensional value sets are likely to produce a large number of combinations of ZEN elements that result in experiments with no useful meaning. The consequence can be a dramatic increase in the time needed to conduct all experiments required by a parameter study. In order to reduce the multi-dimensional value set to a meaningful and interesting subset, we introduce the ZEN constraint directives.

The *ZEN constraint directive* defines a boolean expression over the value sets of several (assign and substitute) ZEN variables. The purpose is to reduce the number of ZEN file instances generated. Depending on the type of the ZEN variables (see Section 3.2) to be included in a constraint, we have to use an appropriate constraint directive: (1) *value set constraint* constrains the value sets of one or more ZEN variables of type integer or real, based on a boolean expression defined over the associated ZEN elements; (2) *index domain constraint* constrains the value sets of one or more ZEN variables of any type (including string) based on a boolean function defined over their index domains (see Section 3.2).

Similar to the substitute directive, ZEN provides constraint directives with global and local scopes. Local ZEN constraint directives can be nested too.

global-constraint	**is**	`CONSTRAINT` *type b-expr*
b-expr	**is**	*bool-expr(zen-var-list)*
type	**is**	`VALUE`
	or	`INDEX`
local-constraint	**is**	`CONSTRAINT` *type b-expr* `BEGIN`
		code-region
		`END CONSTRAINT`

The term *b-expr* refers to a boolean expression with constants and ZEN variables as operands. The set of arithmetical operators allowed in a *b-expr* is: $\{+, -, *, /, \%, \hat{}\}$, the set of relational operators: $\{==, !=, <, >, <=, >=\}$ and the set of logical operators: $\{!, \&\&, ||\}$. Note that $\%$ is the modulo and $\hat{}$ the power operator. For every operator we assume the standard mathematical associativity, which can be overwritten by using parenthesis. Arithmetical operators have precedence over relational operators which have precedence over logical operators. An arithmetical operation over integers produces an integer result, whereas an operation over mixed integers and reals produces a real result.

A ZEN constraint directive d, which defines the boolean expression *bool-expr(zen-var*$_1$*, …, zen-var*$_n$*)*, holds for every ZEN variable in the scope of the directive with the name in { *zen-var*$_1$*, …, zen-var*$_n$ }. This means that if there exist homonym ZEN variables in the scope of the directive with the name in { *zen-var*$_1$*, …, zen-var*$_n$ }, a set of constraints are generated, as follows: $\{\textit{bool-expr}(z_1, \ldots, z_n) \mid \forall\ \{z_1, \ldots, z_n\} \subset \textit{scope(d)},\ \textit{such that}\ \nu(z_i) = \textit{zen-var}_i,\ i \in [1 \mathinner{..} n]\}$, where $\nu(z_i)$ is the textual name of a ZEN variable, defined in Section 3.5.

Let $z_1, \ldots, z_n$ denote a set of ZEN variables. The tuple $(e_1, \ldots, e_n) \in \mathcal{V}^{z_1} \times \ldots \times \mathcal{V}^{z_n}$ is *valid* iff $\forall$ value set constraints *bool-expr(*$z_1, \ldots, z_n$*)* defined, *bool-expr(*$e_1, \ldots, e_n$*) = true*. Similarly, the tuple $(i_1, \ldots, i_n) \in \mathcal{I}^{z_1} \times \ldots \times \mathcal{I}^{z_n}$ is valid iff $\forall$ index domain constraint *bool-expr(*$z_1, \ldots, z_n$*)* defined, *bool-expr(*$i_1, \ldots, i_n$*) = true*.

We now redefine the multi-dimensional value set from Definition 3.4, to take ZEN constraints into account.

Definition 3.5 The *multi-dimensional value set* for a set of ZEN variables $z_1, \ldots, z_n$ is defined as:
$\mathcal{V}(z_1, \ldots, z_n) = \left\{(e_1, \ldots, e_n) \in \mathcal{V}^{z_1} \times \ldots \times \mathcal{V}^{z_n} \mid (e_1, \ldots, e_n)\ \textit{is valid}\ \wedge\ \left(\vartheta^{-1}(e_1), \ldots, \vartheta^{-1}(e_n)\right)\ \textit{is valid}\ \right\}.$

Example 3.6 (Global Value Set Constraints)

```
!ZEN$ ASSIGN N = { 2**{6:12} }
!ZEN$ ASSIGN P = { {2:8:2}**2 }
!ZEN$ CONSTRAINT VALUE N^3 / P < 10000000
```

In Example 3.6, the ZEN variable N defines the powers of 2 ranging from 2^6 to 2^{12} and P the square numbers from 2^2 to 8^2 with stride 2: $\mathcal{V}^N$ = { 2**6, 2**7, 2**8, 2**9, 2**10, 2**11, 2**12 }, $\mathcal{V}^P$ = { 2**2, 4**2, 6**2, 8**2 }. The value set constraint directive filters the ZEN elements from $\mathcal{V}^N \times \mathcal{V}^P$, such that the boolean expressions defined yields true. If we assume that N is the size of a 3-dimensional array and P the number of available processors onto which the array is distributed, the constraint restricts the value set to those combinations which need less than 10MB on each processor.

Example 3.7 (Local Index-Domain Constraints)

```
!ZEN$ CONSTRAINT INDEX INPUT1 == OUTPUT1 BEGIN
!ZEN$ SUBSTITUTE INPUT1 = { INPUT{1:100} }
OPEN(UNIT=2, IOSTAT=IOS, FILE='INPUT1', STATUS='OLD')
!ZEN$ END SUBSTITUTE
...
!ZEN$ SUBSTITUTE OUTPUT1 = { OUTPUT{1:100} }
OPEN(UNIT=2, IOSTAT=IOS, FILE='OUTPUT1', STATUS='NEW')
!ZEN$ END SUBSTITUTE
!ZEN$ END CONSTRAINT
```

Parameter studies [1, 18] are applications that are executed for different input parameters in order to examine their effect on the corresponding output parameters. Usually, output parameter values are written to distinct output files for every experiment. Example 3.7 gives an example how ZEN directives can be employed to manage such parameter studies. The ZEN substitution directives are used to specify all different input data files and all output files, to which the results will be written during an experiment. In order to avoid irrelevant input/output file combinations, we employ ZEN constraint directives to associate every input file with a specific output file: $\mathcal{V}(\texttt{INPUT1}, \texttt{OUTPUT1}) = \{(\texttt{INPUT}i, \texttt{OUTPUT}i) \mid i \in \mathbb{N},\ i \in [1\,..\,100]\}$. Thus, 100 ZEN file instances are generated, each of them reading from and writing to different input and output files, respectively.

3.8 ZEN Performance Behaviour Directive

For performance-oriented program development [16] the user commonly requires information about the performance behaviour for specific code regions, such as the number of cache misses, communication time or floating-point operations per second. ZEN supports the specification of performance metrics to be measured for specific code regions through the *ZEN performance behaviour directive*.

The scope of this directive can be limited to local code regions or the entire file, through the following syntax:

global-measure	**is**	`CR` *cr_mnem-list* `PMETRIC` *pm_mnem-list*
local-measure	**is**	`CR` *cr_mnem-list* `PMETRIC` *pm_mnem-list* `BEGIN`
		code-region
		`END CR`

Two sets of mnemonics can be used to specify code regions (*cr_mnem*) and performance metrics (*pm_mnem*). Overall ZEN supports approximately 50 code regions (e.g. `CR_P` = whole program, `CR_L` = all loops, `CR_OMPPA` = all OpenMP parallel loops) and 40 performance metric mnemonics (e.g. `ODATA` = data movement, `OSYNC` = synchronisation, `ODATA_L2` = number of level 2 cache misses) for various programming paradigms, including OpenMP [3], HPF [6] and MPI [11]. A complete list of mnemonics supported by ZEN can be found in [15]. In order to obtain the performance data for the chosen code regions, an interface to an external performance tool has to be provided (see Section 4).

A global performance directive d collects performance metrics for all code regions in the ZEN file which contains d. The performance metrics are specified in the `PERF` part and the code regions in the `CR` part of d. The local performance directive restricts performance information to the corresponding code region. There can be nested performance measurement directives with arbitrary combinations of global and local directives. If different performance metrics are requested for a specific code region by several nested directives, then the union of these metrics is determined.

Two examples of using the performance behaviour directives are given in Section 5.

4 Prototype Implementation

Figure 6 shows the basic architecture of an automatic experiment generator prototype. The ZEN specification is language independent, except for the ZEN directive prefix. However, ZTS has to deal with several dependencies which include: the syntax of an assignment statement generated for a ZEN assignment directive, the case sensitivity of the string substitution, and the mapping of the ZEN behaviour directives to a performance instrumentation system. ZTS identifies the programming languages in which ZEN files were written based on the file extension. This can be overwritten (or defined, in case of input files) through a command line option. Our implementation of the ZEN performance behaviour directives supports an interface

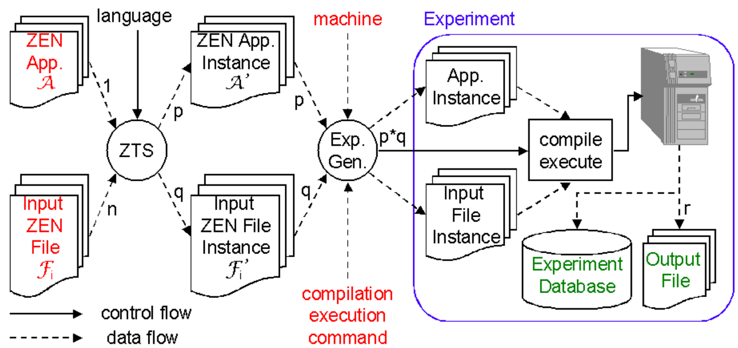


Figure 6. The Experiment Generator Architecture, where $p = |\mathcal{V}^{\mathcal{A}}|$, $q = |\mathcal{V}^{\mathcal{F}_1} \times \ldots \times \mathcal{V}^{\mathcal{F}_n}|$**.**

to the SCALEA [16] automatic performance measurement tool for parallel and distributed Fortran 90 programs. Based on the directives encountered, SCALEA automatically inserts probes into ZEN files to measure the requested performance metrics during an experiment. Of course, any performance tool that knows the semantics of ZEN can be incorporated to obtain performance information. The input to the system is given by a ZEN application $\mathcal{A}$ (comprising all application files, including source files, Makefiles, execution scripts, etc.), a set of ZEN input files $\mathcal{F}_i$, $i \in [1 .. n]$, a compilation and an execution command, and the name of the target site (single machine, parallel system, GRID system, etc.) on which the experiments will be executed. The typical compilation command will be `make`, for which a Makefile must be provided as an application (possibly ZEN annotated) file. The remainder is processed fully automatically by the prototype system. The ZEN application and ZEN input files are processed by ZTS to produce $p = |\mathcal{V}^{\mathcal{A}}|$ application instances and $q = |\mathcal{V}_1^{\mathcal{F}} \times \ldots \times \mathcal{V}_n^{\mathcal{F}}|$ input ZEN file instances. Then the *experiment generator* computes the cartesian product of the set of application and input file instances and generates $p \times q$ different experiments. By using the compilation and execution commands, the experiments are automatically compiled and executed on the target site. A Java-based front-end client displays an on-line view of the execution status (e.g. compiling, queued, running, terminated) of all experiments generated. After the completion of each experiment, the corresponding outputs (i.e. output files and performance data) can be optionally stored into an *experiment database* or at any (`ftp`) location indicated by the user. By using SQL queries, we interogate the database for performance and output data and export the resulting information to Microsoft Excel for visualisation.

5 Experiments

In order to evaluate our prototype experiment generator, we have conducted two different experiments: a performance study for an ocean simulation application (see Section 5.1) and a parameter study for a backward pricing application (see Section 5.2).

5.1 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 [14]. Jobs are scheduled on dedicated nodes using PBS (Portable Batch queuing System [17]). 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. We studied the scalability behaviour and the impact of various OpenMP loop distribution strategies on the resulting performance.

The *scalability* of the code was examined by varying two parameters: (1) the machine size, which consists of two dimensions: (i) the number of SMP nodes, controlled by directives inserted in the PBS script (see Example 5.1); (ii) the number of threads per SMP node, controlled by the input parameter to the `omp_set_num_threads` OpenMP library routine (see Example 5.2); (2) the problem size, by varying the grid (ocean) size and the number of iterations on the grid (see Example 5.3).

Example 5.1 (PBS Script)

```
#!ZEN$ SUBSTITUTE nodes\=2 = { nodes={1:10} }
#PBS -l walltime=1:00:00,nodes=2:fourproc:ppn=4
#PBS -q @gescher.vcpc.univie.ac.at
#PBS -N stommel
#!ZEN$ SUBSTITUTE $4 = { 1:10 } BEGIN
mpirun -np $4 $PBS_O_WORKDIR/omp_02 < $PBS_O_WORKDIR/st.in
#!ZEN$ END SUBSTITUTE
#!ZEN$ CONSTRAINT INDEX nodes\=2 == $4
```

Example 5.2 (Source Code Excerpt)

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

Example 5.3 (Input Data File – `st.in`)

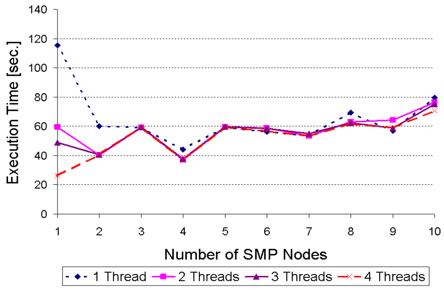
```
!ZEN$ SUBSTITUTE points = { 200, 400 }
      points points
      2000000, 40000000
      1.0e-9 2.25e-11 3.0e-6
!ZEN$ SUBSTITUTE iters = { 20000, 40000 }
      iters
!ZEN$ CONSTRAINT INDEX points == iters
```

Only 8 ZEN directives had to be included in three files of this application to generate the desired 80 experiments: $|\mathcal{V}(\mathit{nodes}{=}2, \$4, \texttt{omp\_set\_num\_threads}(4), \mathit{points}, \mathit{iters})| = 80$. In addition, the compilation and execution command had to be provided. The remainder, including the performance visualizations shown in Figure 7, has been done automatically by the the prototype system. For each experiment the overall execution time and the communication overhead are measured (denoted by the ODATA and WTIME mnemonics in Example 5.2), by using the SCALEA [16] performance instrumentation library. After completing an experiment, the performance data is processed by SCALEA and stored in the experiment database. SQL queries are used to retrieve this data and Microsoft Excel to visualize the performance data (see Fig. 7).

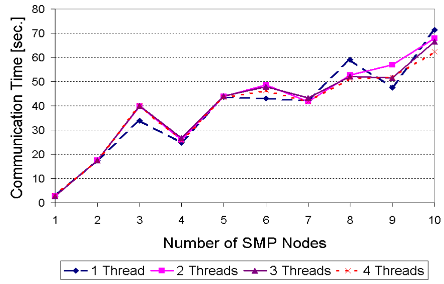
For a 200×200 grid, the program does not scale (see Figure 7(a)) which is explained by the poor communication behaviour (see Figure 7(b)) for this problem size. We also observed that this problem size scales well with the number of threads on a single SMP node. For larger number of nodes the number of threads does not improve the overall performance. The reason is the large MPI communication overhead which dominates the overall execution time.

The 400×400 grid problem size shows a very reasonable scaling behaviour for up to 4 SMP nodes (see Figure 7(c)). 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 computation and communication time (see Figure 7(d)). 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.

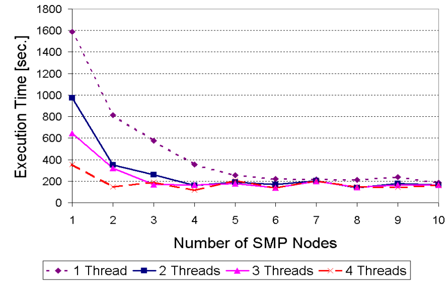
A second experiment was conducted to examine the different OpenMP loop scheduling strategies and their performance effects. Various options are provided to change the work distribution of loop iterations, which includes the scheduling strategy (i.e. STATIC, DYNAMIC and GUIDED) and the size of iterations (chunk) distributed onto the set of threads. Both the scheduling strategy and the chunk size have been varied through the usage of ZEN substitution directives (see Exam-



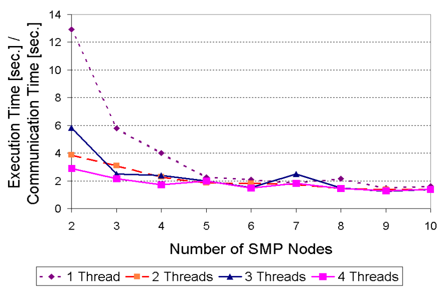
(a) 200 × 200 Grid, 20000 Iterations



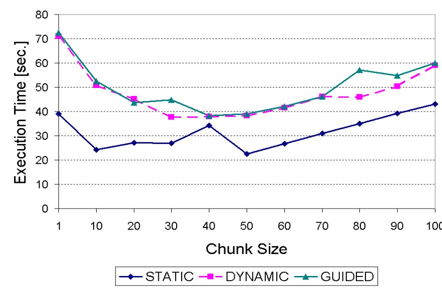
(b) 200 × 200 Grid, 20000 Iterations



(c) 400 × 400 Grid, 40000 Iterations



(d) 400 × 400 Grid, 40000 Iterations



(e) 4 Threads, 20000 Iterations

Figure 7. Performance study for an ocean simulation application with varying problem sizes, machine sizes and loop distribution strategies.

ple 3.2). We requested the execution time (see Figure 7(e)) of OpenMP `PARALLEL` regions by inserting the following ZEN performance behaviour directive:

```
!ZEN$ CR CR_OMPPA PERF WTIME
```

Figure 7(e) shows that for the problem size examined, `STATIC` scheduling performs better than `DYNAMIC` and `GUIDED`. The optimal chunk size is 50. Static scheduling is most likely superior because it implies the least runtime scheduling overhead.

By using ZEN and our prototype experiment generator, 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 used? What scheduling algorithm is best to be used for a parallel loop?

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

The backward pricing kernel [4] 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).

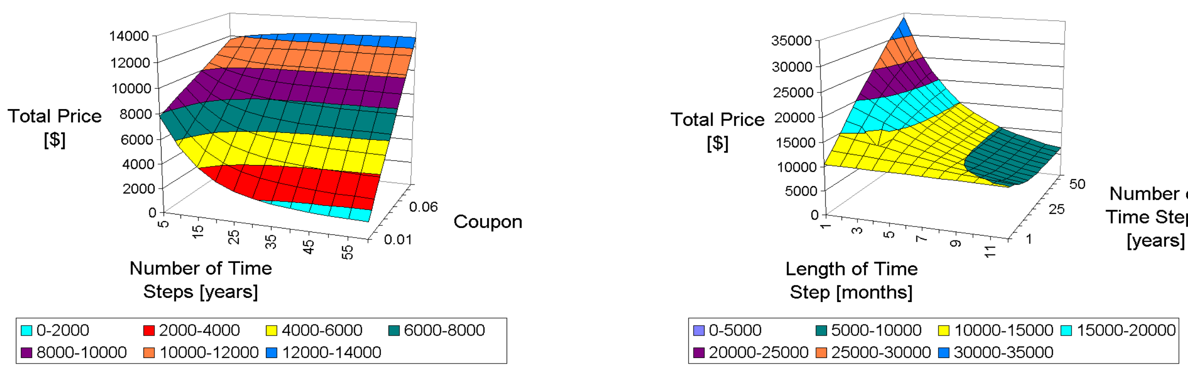
Example 5.4 (Input Parameter Assignment)

```
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 Example 5.4). 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. Note that for this experiment we are not interested in performance aspects, but concentrated 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 Figure 8. The 3D surface in Figure 8(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 $\times$ 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 8(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.



(a) Length of Time Step = 1.0

(b) Coupon = 0.05

Figure 8. Evolution of the total price for the Backward Pricing application.

6 Conclusions and Future Work

We have described ZEN, a new directive based language for automatic experiment management. By inserting ZEN directives into arbitrary files (e.g. program, input and script makefiles) it is possible to specify arbitrarily complex program executions for parameter studies, performance analysis and tuning and software testing, for a wide variety of parallel and distributed architectures. ZEN directives are interpreted as comments and therefore ignored by non-aware ZEN systems, whereas ZEN-aware systems can employ these directives to generate appropriate experiments.

We have implemented a prototype experiment management system that includes a ZEN Transformation System (ZTS), an interface to a performance instrumentation and monitoring system, and an experiment generator and monitor. This system hides from users the intimate details of job scheduling, code instrumentation for performance analysis, transferring codes and input files to various target architectures, application compiling and job launching. Thus, scientists and engineers can concentrate on the science of the underlying experiments. We have demonstrated the usefulness of our prototype implementation on the performance analysis of an ocean simulation application and on the parameter study of a computational finance code.

We are currently in the process of developing a sophisticated graphical user interface for our prototype system, which allows the user to specify job scheduling strategies, target machines, application, input and output file locations, output and performance data visualisations, etc. We also extend our prototype system for larger classes of target systems including GLOBUS [5]. Finally, we will also support fault tolerance to deal with system and application failures.

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, Aug. 1995. IEEE Computer Society Press.

[2] M. Baker and R. Buyya. Cluster computing at a glance. In R. 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] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan./Mar. 1998.

[4] 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.

[5] 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.

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

[7] 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.

[8] Y. E. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. ZOO: A desktop experiment management environment. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, pages 274–285, Mumbai (Bombay), India, 3–6 Sept. 1996. Morgan Kaufmann.

[9] K. L. Karavanic and B. P. Miller. Experiment management support for performance tuning. In ACM, editor, *Proceedings of the SC'97 Conference*, San Jose, California, USA, Nov. 1997. ACM Press and IEEE Computer Society Press.

[10] M. Metcalf and J. Reid. *Fortran 90/95 explained.* Oxfor Science Publications, 1996.

[11] M. P. I. F. MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, January 1996.

[12] R. Reussner, P. Sanders, L. Prechelt, and M. Mueller. SKaMPI: A detailed, accurate MPI benchmark. *Lecture Notes in Computer Science*, 1497:52, 1998.

[13] 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, Aug. 1999.

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

[15] H.-L. Truong and T. Fahringer. Scalea - a performance instrumentation, measurement and analysis tool for cluster and grid computing. www.par.univie.ac.at/project/scalea, Institute for Software Science, University of Vienna.

[16] H.-L. Truong and T. 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.

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

[18] M. Yarrow, K. M. McCann, R. Biswas, and R. F. V. 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, Dec. 2000. ACM Press and IEEE Computer Society Press.