DISSERTATION

Experiment Management, Performance Optimisation, and Tool Integration in Grid Computing

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

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

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl. Ing. Radu Prodan
Matrikelnummer: 0027852
Mariahilfpark 4 / 502, A-6020 Innsbruck

Wien, am 10. August 2004

Abstract

In the past years, the interest in computational Grids has increasingly grown in the scientific community as a mean of enabling the application developers to aggregate resources scattered around the globe for solving large-scale scientific problems. Developing applications that can effectively utilise the Grid, however, still remains very difficult due to the lack of high-level tools to support developers.

For instance, existing available performance analysis tools target single application execution, which is not sufficient for efficient performance tuning of parallel applications. The most popular performance metrics such as speedup or efficiency, for instance, require repeated execution of the application for various machine sizes, for which no automatic tool support exists so far.

The thesis proposes a new directive-based language called ZEN for compact specification of wide value ranges for arbitrary application parameters, including problem or machine sizes, array or loop distributions, software libraries, interconnection networks, or target execution machines. The ZEN directives are problem independent and offer a fine-grained scope that does not change the semantics of the application, nor does it require any application modification or special preparation. Irrelevant or meaningless experiments can be eliminated through a constraint mechanism. Additionally, the ZEN directives can be used to specify a wide range of performance metrics to be collected from the application for arbitrary code regions.

Based on the ZEN language, the thesis proposes a novel experiment management tool called ZENTURIO for automatic experiment management in the context of large-scale performance and parameter studies on the Grid. ZENTURIO offers automatic cross-experiment analysis and visualisation support based on the application performance and output data which are well-organised in a public domain data repository. In contrast to existing parameter study tools, ZENTURIO requires no special preparation of the application and does not restrict the parameterisation to input files or to global input arguments.

ZENTURIO has been designed as a distributed service-oriented architecture based on the latest state-of-the-art Web and Grid services technologies. The thesis illustrates how a service-oriented architecture facilitates the integration of a broad set of tools and enables a range of useful tool interoperability scenarios. A variety of novel Web technology adaptations for Grid computing are presented, which anticipated several standardisation efforts currently still under way within the Global Grid Forum.

ZENTURIO designs an optimisation framework that integrates general-purpose heuristics for solving NP-complete performance and parameter optimisation problems in a wide search space specified using the ZEN language. New optimisation problems can be easily instantiated by simply providing the appropriate objective function, for instance a performance metric using the ZEN language. As a case study, a genetic algorithm is applied on scheduling various types of Grid applications.

The thesis proposes a new hybrid approach for scheduling new classes of workflow Grid applications, which combines static scheduling as an optimisation problem with dynamic steering based on the Grid resource availability. In addition, this is the first scheduling approach that formally handles recursive loops that are often encountered in scientific workflows.

The thesis presents a variety of real-world experiments that validate the research topics addressed.

Acknowledgements

I thank my advisor, Prof. Thomas Fahringer, for the opportunity of working in his research group at the University of Vienna. His support and guidance during my Ph.D. years have been invaluable.

I thank Prof. Blieberger for accepting to be my co-examiner. His comments on the first draft of the thesis helped me improve the presentation in many aspects.

I thank Prof. Zima for the opportunity of working in the AURORA special research program that funded my Ph.D. study. His lecture on Supercompilers remains a milestone in my education.

I thank all the members of the AURORA project for their cooperation which had a deep impact on the outcome of my work. The Klausur has always been an enjoyable opportunity for new interactions and long informal discussions.

John Kewley has been very close to me during the initial years of my Ph.D. work.

Finally, I thank Patrizia for the abundance of pasta, Manuel for the countless hours of Star Wars, and my parents and my entire family for their encouragements during the difficult times that saw me through.

Contents

List of Figures

List of Tables

1

Introduction

Before 1990, the world-wide *Internet* network was almost entirely unknown outside the universities and the corporate research departments. The common way of accessing the Internet was via command line interfaces such as `telnet`, `ftp`, or popular Unix mail user agents like `elm`, `mush`, `pine`, or `rmail`. The usual access to information was based on peer-to-peer email message exchange which made the every day information flow slow, unreliable, and tedious. The advent of the *World-Wide Web* has revolutionised the information flow though the Internet from the obsolete message-passing to the world-wide Web page publication. Since then, the Internet has exploded to become an ubiquitous global infrastructure for publishing and exchange of (free) digital information.

Despite its global success and acceptance as a standard mean of publishing and exchange of digital information, the World-Wide Web technology does not enable ubiquitous access to the billions of (potentially idle) computers simultaneously connected to the Internet providing petaflops of estimated aggregate computational power. Remote access to computational power is highly demanded by applications that simulate complex scientific and engineering problems, like medical simulations, industrial equipment control, stock portfolio management, weather forecasting, earthquake simulations, flood management, and so on.

Nowadays, the common policy of accessing high-end computational resources is through manual remote `ssh` logins on behalf of individual user accounts. Similar to the World-Wide Web that revolutionised the information access, the *computational Grids* are aiming to define an infrastructure that provides dependable, consistent, pervasive, and inexpensive access to the world-wide computational capabilities of the Internet [65]. In this context, computational Grids raise a new class of important scientific research opportunities and challenges regarding, e.g.,:

- secure resource sharing among dynamic collections of individuals and institutions forming so called *Virtual Organisations* [65];

- solving large-scale problems for which appropriate local resources are not available;
- improving the performance of applications by increasing the parallelism through concurrent use of distributed Grid computational resources;
- course-grain composition of large-scale applications from off-the-shelf pre-installed software components;
- exploiting (or stealing) unused CPU cycles from idle (e.g., desktop, PC laboratory) computers to increase the overall compute power;
- intelligent distribution and replication of large data files close places where subsequent computations will take place;
- incorporation of semantic Web technologies [42].

1.1 Motivation

In the past years, the interest in *computational Grids* has increasingly grown in the scientific community as a mean of enabling application developers to aggregate resources scattered around the globe for solving large-scale scientific problems. Developing applications that can effectively utilise the Grid, however, still remains very difficult due to the lack of high-level tools to support developers.

This thesis aims to meet various aspects with respect to integrated tool development for efficient engineering and execution of applications on the Grid.

1.1.1 Performance Tuning

Computational Grids have the potential to harness remote high-performance platforms for the efficient execution of scientific applications. Existing parallel applications that leverage the currently successful parallel programming standards [40, 145], however, require to be tuned to the characteristics of each particular parallel architecture in order to achieve high-performance. The compiler technology has proven to be inefficient in transparently parallelising the applications which still rely on the manual user support. Existing performance tools offer help for advanced analysis of single experiments only, which is not sufficient for efficient application performance tuning.

In a traditional approach, the performance tuning of parallel applications is a multi-experimental cyclic process. The most popular performance metrics, such as efficiency or speedup, require the investigation of numerous problem and machine sizes for, e.g., various compiler options and data or control flow distributions. This process involves many cycles of code editing, compilation, execution, data collection, performance analysis, and data visualisation, which is tedious and error-prone to be managed manually. To this date there is no support for automatic cross-experiment performance analysis of parallel applications.

1.1.2 Parameter Studies

In the last decade, large-scale parameter studies have become feasible through the appearance of parallel compute engines with multi-gigabyte memories and terabyte disk farms. Such parameter studies require repeated invocation of the same application on a variety of input data sets combined with appropriate organisation of the output data files for subsequent analysis and visualisation. Existing parameter study tools like Nimrod [2] or ILAB [178] require special preparation of the application, which is usually the main obstacle for a tool in achieving wide acceptance. The application developers are in general very reluctant in changing their applications to the peculiarities of each tool and prefer to write special purpose scripts hard-coded for their specific parameter studies, instead of using general purpose tools that can give them enhanced graphical interfaces and fault tolerance support.

1.1.3 Optimisation

Exhaustive performance and parameter studies describe the complete evolution of the performance metric or the output parameter under evaluation as a function of the indicated input parameters. While such studies provide invaluable information on the application behaviour, they often produce an overflow of data which is irrelevant for further studies. In many cases parameter spaces become so large that they are impossible to be exhaustively traversed. Often the users are only interested in finding parameter combinations that optimise a certain performance metric or an output parameter, rather than conducting the complete set of experiments for all parameter combinations. This is typically an NP-complete problem that requires heuristic-based approaches. The performance tuning and the scheduling of applications are two such typical NP-complete optimisation tasks. There are currently no tools to support the users in defining and solving general NP-complete optimisation problems for scientific applications on the Grid.

1.1.4 Scheduling

Fine-grained performance analysis and tuning, as is usually performed on traditional parallel computers, is often unrealistic to be applied to world-wide course-grain computational Grid infrastructures. The problem of high-performance execution of scientific applications gets shifted from fine-grained performance analysis and tuning to appropriate scheduling onto the available computational Grid resources.

Application scheduling in a classical approach is an NP-complete optimisation problem [165]. The scheduling search space which exponentially depends on the (potentially unbounded) number of resources and tasks and can achieve particularly huge dimensions on the Grid which have not been previously addressed. In addition, the static scheduling as an optimisation problem has to

be enhanced with steering capabilities that consider the dynamic availability of the Grid resources over space and time.

The workflow model originating from business process modelling [173] is gaining increased interest as the potential state-of-the-art paradigm for programming Grid applications. While business process workflows are in most cases Directed Acyclic Graphs (DAG) that consist of a limited number of nodes, scientific workflows that implement Grid applications often require large iterative loops that implement a convergence behaviour or a recursive problem definition. There is currently no systematic formal approach to scheduling workflow Grid applications that combines classical DAG optimisations with recursive loop handling.

1.1.5 Parametrisation Language

One reason why there is no tool support for automatic experiment management regardless the ultimate goal (i.e., performance studies, parameter studies, optimisations) is the lack of appropriate languages to define experiments. Currently each user takes own ad-hoc approaches in defining value ranges for relevant application parameters by writing hard-coded scripts that serve a very specific experimental purpose. Moreover, existing performance and parameter study tools [4] that offer some support for automatic experiment management, approach the parameter specification problem in a similar ad-hoc manner through special-purpose external scripts that force the developers to export application parameters to external global variables. Other tools [178] that aim for a more flexible parameter specification through graphical annotations are restricted to input files.

The limitations of the existing parameter specification approaches can be summarised as follows:

1. the parameter specification is restricted to input files or program arguments;
2. only global variables or program arguments can be expressed;
3. local variables cannot be parameterised;
4. parallelisation strategies (e.g., array and loop distributions) or other application characteristics cannot be expressed;
5. the parametrisation forces the user to perform undesired modifications and adaptations of the application;
6. there is no formal approach to define a general-purpose experiment specification language.

1.1.6 Instrumentation

Program instrumentation is a common task that all the performance analysis tools need to perform for measuring and collecting run-time application data. The instrumentation technologies developed so far have the following drawbacks:

1. *source code instrumentation* [12, 134] forces the user to manually insert probes in the application which, apart from being tedious to perform, often introduces undesired source code modifications that are bound to the profiling library used;
2. *compiler instrumentation* through external flags as performed by most commercial compilers has serious limitations in the specification of fine-grained local source code regions for which to the collect performance data;
3. *dynamic instrumentation* [24] and
4. *binary rewriting* [82] do not perturbate the original source code, but are limited to binary executables, impossible to be reversibly mapped to the original source code. In addition, the portability of these technologies is very critical;
5. *object code wrapping* [23] is limited to pre-compiled software libraries.

1.1.7 Portability

The set of tools available on each individual platform is usually heterogeneous in functionality and the user interface provided. Before using a new parallel system, the users must in most cases learn and familiarise themselves with new tools with different functionality and user interfaces. This requires (often unnecessary) extra time and effort and can be a major deterrent against using more appropriate computer systems. The main reason for tools not being available on a large set of platforms is their limited portability.

1.1.8 Tool Interoperability

The cooperative use of software tools can significantly improve the application engineering process. For instance, an experiment management tool can make use of a performance monitor for cross-experiment performance analysis and tuning. Or else, the use of on-line performance tools in conjunction with correctness debuggers can significantly improve the performance steering process by applying on-the-fly program modifications based on the on-line performance data analysis.

Unfortunately, most of the tools supporting different phases of the application engineering process cannot be used in cooperation to further improve the user efficiency, because they are insufficiently integrated into a single coherent environment. The main reason for the lack of interoperability between tools are the incompatible monitoring systems and the critical (not isolated) platform dependencies. Each tool requires special preparation of the application which is in most cases the main incompatibility cause.

1.1.9 Stateful Grid Services

The Grid community has acknowledged the Web services [170] as the fundamental technology for building service-oriented infrastructures for the Grid.

The stateless Web services standards designed for business process modelling have, however, fundamental limitations in modelling Grid resources that are by definition stateful. While there are present approaches that aim to define new standards for modelling stateful resources with Web services [163, 67], there are little efforts that analyse and validate their appropriateness.

1.2 Goals

This thesis addresses the motivating problems outlined in the previous section in the context of a novel experiment management tool developed in the frame of an open architecture for tool development on the Grid.

1.2.1 ZEN Directive-based Language

The thesis proposes a new *directive-based language* called ZEN [123, 127] for the specification of arbitrary application parameters through annotations of arbitrary application files. The so called ZEN directives are language independent comments with a well-defined syntax that do not change the semantics of the application source files, as they are ignored by the compilers that are unaware of their semantics. The scope of the ZEN directives can be global or restricted to arbitrary code regions, which allows local fine-grained parameterisation. Simple macro-processor-based string replacement semantics of the ZEN directives insure that the language is not specific to any particular scope and can express new problems that were not thought during the language design. Invalid experiments can be filtered through parameter constraints. Performance directives are introduced to specify the metrics to be measured and computed for fine-grained code regions, without altering the application source code with instrumentation probes. The directive-based approach ensures flexible parameterisation that does not require any source code modification or adaptation.

1.2.2 ZENTURIO Experiment Management Tool

The thesis proposes a novel general-purpose *experiment management tool* called ZENTURIO [124, 128] applied to *performance and parameter studies* of parallel and Grid applications. ZENTURIO uses the ZEN directive-based language to define potentially large value ranges for arbitrary application parameters, including program variables, file names, compiler options, target machines, machine sizes, scheduling strategies, or data distributions, without intruding in the source code or force the application developer to perform any modifications. A graphical User Portal enables the user to easily create, control, and monitor large sets of experiments. An Experiment Generator service parses application files annotated with ZEN directives and generates

synthetic experiments based on the semantics of the directives encountered. An Experiment Executor service retrieves a set of experiments and automatically compiles, executes, and monitors them on the target machine. Upon the completion of each experiment, the output files and performance data are automatically stored into an Experiment Data Repository for post-mortem multi-experiment performance and parameter studies. An Application Data Visualiser portlet of the User Portal has been designed to automatically query the database for performance and output data required for user analysis. A wide set of diagrams [57] are provided to visualise the variation of any performance metric or output parameter as a function of arbitrary ZEN-annotated parameters.

1.2.3 Optimisation

ZENTURIO provides a *modular framework* for solving customisable *performance and parameter NP-complete optimisation problems* to be flexibly instantiated by the user [129].

1. The optimisation problem is specified by providing an *objective function* that must implement a well-defined *problem independent interface.* As case studies, ZENTURIO instantiates the objective function for three optimisation problems:
   a) application-specific analytical prediction function for single static workflow scheduling;
   b) random function for simulated independent task-set scheduling;
   c) performance metric for performance tuning of parallel applications. The performance metric is specified by a ZEN performance directive and measured through experiment execution;
2. *General purpose heuristics* are employed to surf the search space defined through ZEN directives for an experiment that maximises the objective function. ZENTURIO illustrates a generic encoding of the heuristic search engine based on *genetic algorithms* and targets various others (including subdivision, simplex, simulated annealing, BFGS, or EPSOC methods) as future work.

1.2.4 Dynamic Workflow Scheduling

The workflow model has emerged as the potential state-of-the-art paradigm for programming Grid applications. On the other hand, the static scheduling as an NP-complete optimisation problem is not enough for efficient execution of applications in a dynamic Grid environment, where resources often change load and availability. Existing ad-hoc approaches either do not address the workflow scheduling as an optimisation problem or are restricted to Directed Acyclic Graph-based workflows that cannot handle *loops.* This thesis proposes

a novel *hybrid approach for dynamic scheduling of Directed Graph-based workflow applications* that dynamically adapts the optimised static schedule to the heterogeneous and changing Grid resources.

1.2.5 Service-oriented Grid Architecture

The main reason why each computing platform has its own heterogeneous set of tools is their limited portability. In addition, the tools are designed as stand-alone and cannot be used in cooperation to improve the user efficiency in the application engineering process. The thesis addresses the portability and interoperability issues through a distributed *multi-layered service-oriented architecture* [97, 98] with the following design principles:

1. The platform dependencies are isolated within stand-alone distributed services and sensors exporting a platform independent API. The client end-user tool is therefore decoupled from the intimate hardware and operating system dependencies which significantly increases the tool portability;
2. A set of general-purpose services for the Grid have been identified and realised;
3. The recommendation that every platform vendor implements a core set of tool services with a platform independent API significantly eases the tool development and multi-platform availability;
4. The functionality of each tool is no longer implemented by a single monolithic tool that acts as a big black-box. Enabling light-weight portals easily to be installed and managed on local client machines significantly simplifies the Grid usage;
5. The services are designed such that they can be concurrently accessed by multiple clients. This enables multiple tools interoperate by sharing the common services which possibly monitor the same target application processes;
6. An asynchronous event framework enables the services to notify the clients about interesting application and system events. Events are important for detecting important status information about the system and the application and can be used to avoid expensive continuous polling.

Beyond the provision of an open framework for tool development, the thesis presents various practical scenarios how *interoperable use of software tools* can significantly improve the productivity in the application engineering process [98, 132].

1.2.6 Stateful Grid Services

The thesis contributes with several proposals for *enhancement and adaptation of the Web services technology* for implementing services that model *stateful Grid resources* [125, 128]:

1. definition and implementation of the Factory design pattern for on-the-fly service instantiation on remote sites;
2. design and implementation of a Registry service for high-throughput service lookup;
3. definition of service compatibility for functionality-based service discovery;
4. adaptation of existing standards for publication of transient service implementations;
5. service lifetime modelling;
6. comparative analysis and benchmarking of existing service-oriented Grid architectures [126, 128].

1.3 Outline

Chapter 2 presents the Grid architectural model which represents the foundation on top of which the concepts presented in this thesis are developed.

Chapter 3 presents a complete formal specification of the ZEN directive-based language used to specify application parameters and performance metrics.

Chapter 4 is devoted to a detailed description of the ZENTURIO experiment management tool, with particular focus on the tool functionality.

Chapter 5 describes the open service-oriented architecture for interoperable tool development, in the frame of which the ZENTURIO experiment management tool has been designed. The set of sensors, the Grid services, the event framework, and several prototype on-line tools, together with various tool interoperability types and scenarios are presented in detail.

Chapter 6 presents the ZENTURIO optimisation framework validated by three case studies: workflow scheduling, throughput scheduling, and performance tuning of parallel applications.

Chapter 7 illustrates practical experiments performed on real-world applications in all the fields addressed by the ZENTURIO experiment management tool: performance studies, parameter studies, and scheduling as an optimisation problem.

Chapter 8 outlines the most relevant related work in all the fields touched by the thesis: experiment management, performance studies, parameter studies, tool interoperability, and scheduling.

Chapter 9 summarises the thesis contributions and gives an outlook to the future research.

2

Model

2.1 Introduction

The mostly used attempt to define Grid computing [65] is through an analogy with the electric power evolution around 1910. The truly revolutionary development was not the discovery of electricity itself, but the electric power grid that provides standard, reliable, and low-cost access to the associated transmission and distribution technologies. Similarly, the Grid research challenge is to provide standard, reliable, and low-cost access to the relatively cheap computing power available nowadays.

Definition 2.1. *A* computational Grid *was originally defined as a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [65]. With the time, the Grid concept has been refined and better formulated, e.g., as a persistent infrastructure that supports computation-intensive and data-intensive collaborative activities that spawn across multiple* Virtual Organisations (VO).

The natural starting point in building computational Grids is the existing world-wide Internet infrastructure that aggregates a potentially unbounded number of resources. Analogous to the World-Wide Web that provides ubiquitous access to the information over the Internet, the computational Grids explore new mechanisms for ubiquitous access to computational resources and quality of service beyond the best-effort provided by the Internet protocol (IP).

There are two recognised architectural approaches for building large-scale Grid infrastructures:

1. *Service-Oriented Architectures (SOA)* [78] are based on an aggregation of portable and reusable programs called services that can be accessed by remote clients over the network in a platform and language independent manner.

Definition 2.2. *A* service *is a self-contained entity program with a well-defined platform and language independent interface that does not depend on the context or the state of other services.*

A service-oriented architecture offers significant advantages on the Grid:
a) it increases the *portability* and facilitates the *maintenance* of the system, by isolating platform dependent services to appropriate sites accessible under a well-defined platform independent API;
b) it enables *light-weight clients* which are easy to be installed and managed by unexperienced users;
c) it *decouples* the clients from the rest of the system and allows the users to *move*, *share*, and *access* the services from different Grid locations.

2. *Peer-To-Peer Architectures (P2P)* [120] are an aggregation of equivalent programs called *peers* situated at the edges of the Internet that provide functionality and share part of their own hardware resources (e.g., processing power, storage capacity, network link bandwidth, printers) with each other through network contention without passing through intermediate entities. The strength of peer-to-peer architectures is the high-degree of *scalability* and *fault tolerance.*

The tool development and integration framework presented in this thesis is build on the foundation of a service-oriented architectural model that is the scope of the remaining part of this chapter.

2.2 Distributed Technology History

The realisation of service-oriented architectures for building distributed Grid infrastructures is the outcome of a long track of research and industry experience on distributed services and component technologies.

Distributed applications require a protocol which defines the communication mechanism between two concurrent remote processes. Traditionally, there have been two communication protocol models for building distributed applications: message passing/queuing and request/response. While both messaging and request/response models have their individual advantages, either one can be implemented in terms of the other. For example, messaging systems can be built using lower-level request/response protocols, which was the case of the Microsoft's *Distributed Computing Environment (DCE)* [139]. For the (Sun) *Remote Procedure Call (RPC)* [144] applications, the synchronous request/response design style is usually a natural fit.

In the 1980s, the communication protocol models focused on the network layer, such as the *Network File System (NFS)* [27] developed originally by Sun Microsystems (which most networked Unix systems currently use as their distributed file system) and Microsoft DCE RPC applications on Windows NT.

In the 1990s, the object-oriented community pushed for an *Object RPC (ORPC)* protocol that links application objects to network protocols. The primary difference between ORPC and the proceeding RPC protocols is that ORPC codifies the mapping of a communication end-point to a language-level object. This mapping allows the server-side middleware locate and instantiate a target object in the server process. The *Common Object Resource Broker Architecture (CORBA)* [107] designed by the *Object Management Group (OMG)* and the Microsoft's *Distributed Component Object Model (DCOM)* [21] have dominated and competed for many years for an ORPC protocol industry standard. Although CORBA and DCOM have been implemented on various platforms, the reality is that any solution built on these protocols are largely dependent on a single vendor implementation. Thus, if one were to develop a DCOM application, all the participating nodes in the distributed application would have to be running a flavour of Windows. In the case of CORBA, every node in the application environment would need to run the same *Object Request Broker (ORB)* product. While there are cases when CORBA ORBs from different vendors do interoperate, that interoperability does not extend into higher-level services such as security and transaction management. Furthermore, any vendor specific optimisations in this situation is lost.

Other efforts such as the *Java Remote Method Invocation (RMI)* [80] from Sun Microsystems enhanced with the *Jini* [50] network awareness are bound to the Java language and fail to fulfill the language independence required by the Grid computing. The *Enterprise Java Beans (EJB)* [140] server-side component technology for the Java 2 Enterprise Edition (J2EE) platform failed to become a standard due to incompatible data formats, limited network transport layer security, the use of non-Web-based communication protocols, and the lack of semantic information in the data representation.

2.3 Web Services

In the year 2000, a consortium of companies comprising Microsoft, IBM, BEA Systems, and Intel defined a new set of XML (eXtensive Markup Language) [81] standards for programming Business-to-Business (B2B) applications called *Web services* [78], which are currently being standardised under the umbrella of the World Wide Web Consortium (W3C) [170]. The motivation behind the Web services is to solve existing barriers between traditional Enterprise Java Beans businesses collaborating in electronic transactions such as incompatible data formats, security issues, Web access, and semantic information. Web Services are a technology for *deployment* and *access* of business functions over the Web that compliments existing standards like J2EE, CORBA, DCOM, RMI, or Jini, which are technologies for *implementing* Web Services.

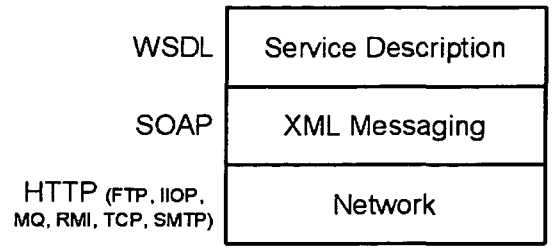


Fig. 2.1. The interoperable Web services stack.

Definition 2.3. *A* Web service *is an interface that describes a collection of operations of a service (see Definition 2.2) that are network-accessible through standardised XML messaging.*

2.3.1 Web Services Stack

The interoperability between Web services is based on a three layer *Web Services Stack* [100], depicted in Figure 2.1.

1. *The Hyper Text Transfer Protocol (HTTP)* is a bottom simple and firewall-friendly RPC-like protocol that is the current de-facto standard for Web communication over TCP/IP;
2. *The Simple Object Access Protocol (SOAP)*[1] [141] is the XML-based message passing standard for communication between remote Web services using both message passing and request/response communication models on top of HTTP. SOAP is open to additional underlying network protocol bindings beyond HTTP, such as CORBA IIOP (Internet Inter ORB Protocol), FTP (File Transfer Protocol), MQ (Message Queuing), RMI, or SMTP (Simple Mail Transfer Protocol). However, in contrast to the popular belief, Web services do not mandate the use of SOAP for Web services communication;
3. The *Web Service Description Language (WSDL)* [31] is the XML standard for the specification of Web services interfaces, analogous to the CORBA Interface Definition Language (IDL). A WSDL document is commonly divided into two distinct parts [100]:
   a) *service interface* is the abstract and reusable part of a service definition, analogous to an abstract interface in a programming language, that can be instantiated and referenced by multiple service implementations. A service interface consists of the following XML elements:
      i. `wsdl:types` contains the definition of complex XML Schema Datatypes (XSD) [171] which are used by the service interface;

[1] This naming is a mistake because the protocol has nothing to do with accessing objects.

ii. `wsdl:message` defines the data transmitted as a collection of logical parts (`wsdl:parts` – e.g., input arguments, return argument, and exception messages), each of which being associated with a different type;
iii. `wsdl:operation` is a named end-point that consumes an input message and returns an output message and a fault message (corresponds to a Java class method);
iv. `wsdl:portType` defines a set of abstract operations (corresponds to a Java interface definition);
v. `wsdl:binding` describes the protocol and the data format for the operations of a `portType`;

b) *service instance*[2] part of a WSDL document describes an instantiation of a Web service. A Web service instance is modelled as a `wsdl:service`, which contains a collection of `wsdl:port` elements (i.e., usually one). A `port` associates one network endpoint (e.g., URL) with a `wsdl:binding` element from a service interface definition.

A common practice is to define the service interface in a separate *abstract interface WSDL document* which is further included into the *instance WSDL document* through an `import` element.

2.3.2 Web Services Publication

The *Universal Description, Discovery and Integration (UDDI)* [164] is a specification for distributed Web-based information registries of business Web services. The WSDL interface and the URL address of persistent Web services are typically published in a centralised UDDI service repository for remote discovery and access. The UDDI best practices document [35] requires that the interface part of the WSDL document be published as a UDDI `tModel` and the instance part as a `businessService` element (i.e., as URLs – see Figure 2.2). The `businessService` UDDI element is a descriptive container used to group related Web services. It contains one or more `bindingTemplate` elements which contain information for connecting and invoking a Web service. The `bindingTemplate` contains a pointer to a `tModel` element which describes the Web service meta-data. An `accessPoint` element is set with the SOAP address of the service `port`.

The *Web Services Inspection Language (WSIL)* [10] defines a distributed Web service discovery method, which is complementary to the UDDI centralised approach. A WSIL document is an XML file that contains references to Web services, which are URLs to instance WSDL documents.

2.3.3 Web Services Security

The *Web Services Security Language (WS-Security)* [9] describes enhancements to the SOAP messaging that provide quality of protection through

[2] The service implementation term used by IBM in [100] is to our opinion wrong.

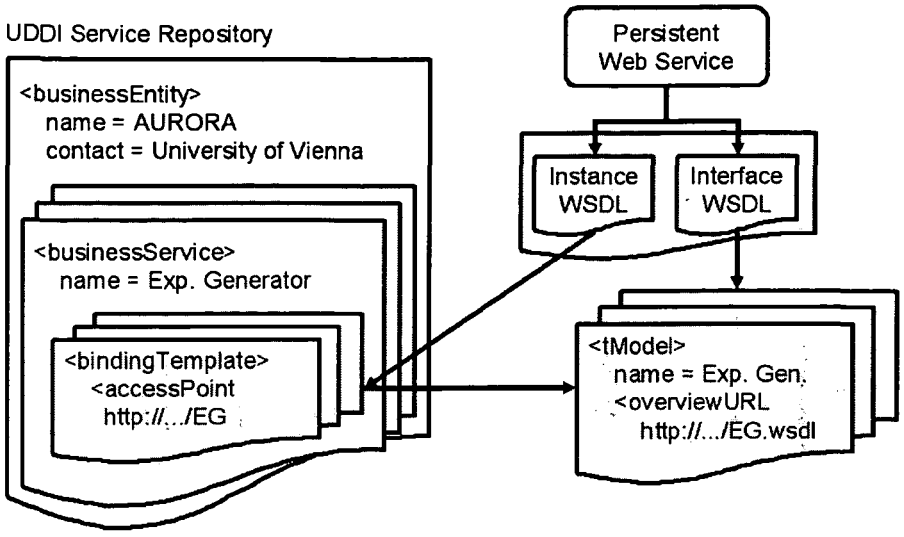


Fig. 2.2. Publishing a Web service into a UDDI service repository.

message integrity (through XML digital signature), message confidentiality (through XML encryption), and single message authentication. These mechanisms can be used to accommodate a wide variety of security models and encryption technologies, including the Public Key Infrastructure (PKI) [13] (see Section 2.4).

2.3.4 Web Services Run-time Environment

The Web services technology omits on purpose to specify any run-time environment that implements the service-oriented architecture based on XML document exchange. Java is currently the most popular programming language supported by high-level Web services implementations due to its platform independent interpreted object code design. Figure 2.3 illustrates the most common run-time architectural model implemented by existing Web services toolkits for Java [51, 59, 70, 114, 153].

Following the CORBA RPC-based model, advanced implementation toolkits completely shield the client application from the underlying XML-based technologies. Existing tools transform / generate the WSDL description of the Web service into / from a (Java) interface definition which is understood by the (Java) clients. Automatically generated *proxies* that export the Web service interface in the client implementation language perform automatic parameter marshaling and (SOAP) message routing.

The Java implementation of the SOAP-based communication infrastructure can be based either on the synchronous JAX-RPC (Java API for XML-based RPC), or on the asynchronous JAXM (Java API for XML messaging)

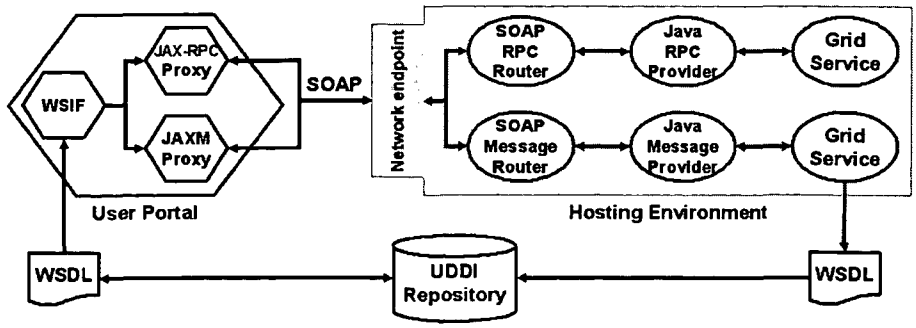


Fig. 2.3. Web services runtime environment.

standard APIs designed by Sun Microsystems. In this context, each remote call from a Java client to a Web service is mapped onto one SOAP JAX-RPC / JAXM message. Additionally, the *Web Services Invocation Framework (WSIF)* [48] allows the invocation of WSDL-described services independently of the underlying (SOAP) protocol implementation.

Similarly to the Enterprise Java Beans component model, Web services typically run within a *hosting environment*, such as the Java 2 Enterprise Edition (J2EE) [58], JBoss, Tomcat [106], Sunl, Weblogic [118], or Websphere [11], which is an HTTP server and servlet engine responsible for deploying and managing the service lifecycle. The Web service functionality is encoded using a Java class that implements the service WSDL interface and deployed using the hosting environment specific tools. Upon receiving a message at the network endpoint of the hosting environment, a SOAP RPC / Message router (servlet) unmarshals the message and forwards it to a Java RPC / Message provider. The Java provider loads the Java class specified in the SOAP message (if not already loaded) that implements the Web service and invokes the appropriate method. The results of the method are returned to the SOAP router which marshals and transfers them to the requesting client.

2.4 Grid Security Infrastructure

The Grid architectural model described in this chapter implicitly assumes the use of the *Grid Security Infrastructure (GSI)* [68] as the de-facto standard for authentication and secure communication across the applications and the services over the Internet. GSI has the following main characteristics:

1. *Public Key Cryptography* [13] based on private and public key pairs is the fundamental technology used for encrypting and decrypting messages;
2. *Digital Signatures* are employed for insuring data integrity over the network;

3. *X.509 Certificates* are used for representing the identity of each Grid user required for authentication. An X.509 certificate includes four primary pieces of information:
   a) *Subject Name* which identifies the person or the object that the certificate represents;
   b) *Public Key* that belongs to the subject;
   c) *Certificate Authority (CA)* that has signed the certificate which certifies that both the public key and the subject name belong to the same trusted subject;
   d) *Digital Signature* of the named certificate authority;
4. *Mutual Authentication* insures that the two parties involved in communication trust each other certificate authorities;
5. *Secure Private Keys* promote the encrypted store of the user private key exclusively on the local personal computer (i.e., laptop) or on cryptographic smartcards;
6. *Single Sign-On* restricts the user authentication to one single password (keyboard) specification during a working session;
7. *Proxy Cryptography* creates a new private and public key pair digitally signed by the user, that temporarily represents the user Grid identity. This allows the true private key of the user be uncrypted for a minimum amount of time, until the signed proxy is generated;
8. *Delegation* allows remote services to behave on behalf of the client through the creation of remote proxies that impersonate the user (see Figure 2.4).

The GSI cryptography can be applied at two layers in the service-oriented Grid architecture proposed in this chapter:

1. *network layer* for communicating with remote light-weight sensors over the Secure Socket Layer (SSL) protocol;
2. *message layer* for secure communication across Grid services based on the Web Services Security Language for signing and encrypting XML SOAP messages, as introduced in Section 2.3.4.

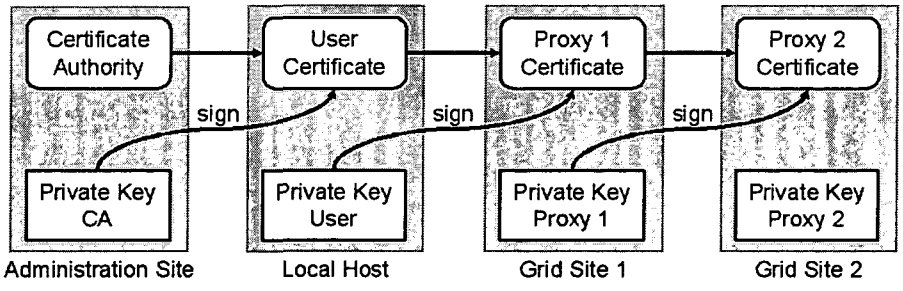


Fig. 2.4. The GSI single sign-on and proxy delegation chain of trust.

Security at the message layer is more powerful than the security at the network layer due to the data encryption at a higher level of abstraction (i.e., it is easier to read a credit card number from an ASCII SOAP message than from a network packet). A higher degree of security can be achieved through authentication and data encryption at both network and message layers, however, at accumulated security overhead costs.

2.5 Globus Toolkit

Since 1995, the *Globus Toolkit (GT)* [64] is developing middleware technology aimed to support and ease the development of high-level Grid infrastructures and applications with special focus on high-performance scientific computing.

The Globus Toolkit Version 2 (GT2), which has been the most successful and stable Globus release at the time this research has been carried out, provides the following three categories of fundamental services for building Grid infrastructures:

1. *Resource Management Services* for executing applications on remote Grid sites, which comprise:
   a) *Globus Resource Allocation Manager (GRAM)* [38] that provides a single standard interface for allocating and using remote computing resources on top of existing job schedulers like Condor [108], Load Sharing Facility (LSF) [179], Maui [34], Portable Batch System (PBS) [166], Sun Grid Engine (SGE) [152], or simple Unix `fork`;
   b) *Dynamically-Updated Request Online Coallocator (DUROC)* [39] that employs multiple GRAM services for multiple Grid site resource coallocation. The lack of resource reservation functionality is the main limitation that hinders DUROC of being largely and effectively used in realistic Grid environments;

   GRAM and DUROC use the *Resource Specification Language (RSL)* to formulate resource requirements;
2. *Information Services* represented by the *Monitoring and Discovery Service (MDS)* [61] that comprises:
   a) *Grid Resource Information Service (GRIS)* that provides information about a particular Grid resource using an underlying sensor (like the Network Weather Service (NWS) [176] for CPU and network information);
   b) *Grid Index Information Service (GIIS)* that provides hierarchical means of aggregating GRIS services for a coherent Grid system image and efficient high-performance resource query support;
3. *Data Grid Services* represented by the:
   a) *Global Access to Secondary Storage (GASS)* [18] libraries and utilities which simplify the porting and running of applications in a Grid environment by installing a transparent distributed file system that eliminates the manual login to remote Grid sites;

b) *GridFTP* [5] which is a high-performance, secure, reliable data transfer protocol optimised for high-bandwidth wide-area networks, based on the highly-popular Internet FTP protocol;
c) *Globus Replica Catalogue* [149] which is a mechanism for maintaining a catalogue of data-set replicas;
d) *Globus Replica Management* [149] which is a mechanism that ties together the Replica Catalogue and the GridFTP technologies for remote management of large data-set replicas.

The Globus Replica Catalogue and the Globus Replica Management are services oriented towards the Data Grid and therefore excluded from the computational Grid architectural model presented in this chapter.

Despite its enormous success in the user Grid research community, GT2 on its own suffers from substantial integration and deployment problems, which is mostly due to the C language-based implementation platform. The Java Commodity Grid Kit (CoG) [168] adds a layer on top of GT2 that exports a platform independent Java interface to the Globus services. GT2 and Java CoG, augmented with GSI and Web services support represent an excellent starting point for implementing higher-level Grid architectures, like the model described in this chapter.

2.6 Grid Architectural Model

Figure 2.5 illustrates a three-tier service-oriented architecture which represents the foundation for the tool development and Grid integration scope of this thesis.

1. *The Machine Layer* is represented by the set of *computational resources*, also called for brevity reasons *machines*, interconnected through the conventional Internet technology that builds in aggregation the physical (hardware) Grid.

   Definition 2.4. *The set of computational resources managed by one hosting environment and one single Globus Resource Allocation Manager service (introduced in Section 2.5) is called* Grid site.

   The machine layer is augmented with a thin set of monitoring *sensors* that may run on every single Grid machine.

   Definition 2.5. *A* sensor *is a small light-weight background program, often also referred as* daemon, *that monitors and collects low-level intimate information about running processes and the underlying computational resources. It additionally exports and provides remote access to this information by means of a well-defined platform independent API.*

   Isolating platform dependencies within sensors under a portable API reduces the effort of porting n services onto m platforms from $n \times m$ to $n + m$.

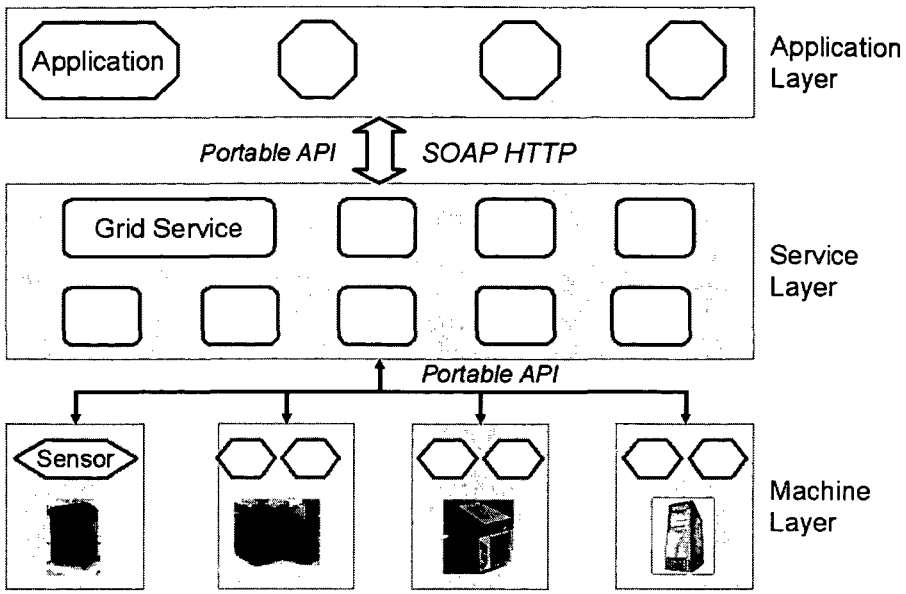


Fig. 2.5. The Grid architectural model.

2. *The Grid Services Layer* largely consists of a set of distributed services that provide generic high-level functionality for advanced tool development, composition, integration, and interoperability. In contrast to the Web services designed to model persistent and stateless business processes, the Grid services need to model transient and stateful Grid resources.

   Definition 2.6. *A* Grid service *is a Web service enhanced with standard interface support for expressing* lifecycle, state, *and* asynchronous events *required for modelling and controlling* dynamic, stateful, *and* transient *Grid resources.*

   A Grid site can host multiple Grid services that can be remotely accessed using Web services XML-based document exchange. In this model there are two persistent Grid services that are required to exist in a Grid environment:
   a) *Factory* for creating transient Grid service instances on arbitrary remote Grid sites;
   b) *Registry* for flexible up-to-date management and high-throughput discovery of transient Grid services.
3. *The Application Layer* is represented by the end-user applications or software tools, built through the course grain *workflow orchestration* (see Section 2.8.2) of the underlying Grid services. The Grid software tool applications are typically represented by graphical user portals or simple batch script front-end programs.

2.7 Stateful Grid Services

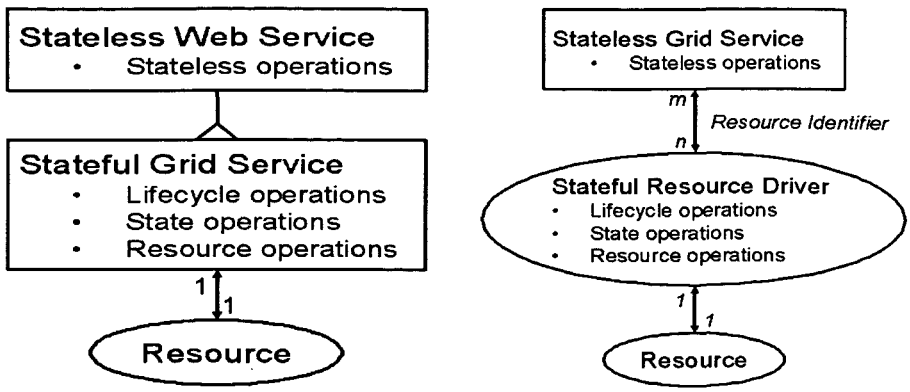
The Grid community has generally acknowledged the Web services as the de-facto standard technology for the realisation of the service-oriented Grid architectures. The *Open Grid Services Architecture (OGSA)* [62] is the generic broad architectural model currently being defined within the Global Grid Forum [30] that defines design mechanisms to uniformly expose Grid services semantics, to create, name, and discover transient Grid service instances, to provide location transparency and multiple protocol bindings for service instances, and to support integration with underlying native platform facilities. Extensive efforts in both Grid [163] and Web [67] communities currently attempt to define a widely accepted standard for building OGSA-compliant interoperable Grid services.

The Grid service concept in Grid computing is associated with the idea of modelling *stateful resources*, which translates into the ability of providing three extensions that are not covered by the standard Web services technology: *lifecycle*, *state*, and *asynchronous events*. Examples of target stateful resources include executing applications, data repositories, Factories for creating Grid service instances, or Registries of existing service instances.

The current Web services standards are purposely focused on *stateless service* modelling and do not intend to specify any standard means for expressing the service state. While there have been several attempts in the Grid community that aimed to standardise the specification of state within Grid services [163, 67], there has been no widely accepted standard by the time the work presented in this thesis has been carried out.

There can be distinguished two orthogonal alternatives of modelling state within Grid services:

1. *Encapsulation* uses the Java Beans model of accessing and manipulating the service state through `get` and `set` interface methods. In this model illustrated in Figure 2.6(a), a *stateful Grid service* specialises the stateless Web service with methods concerning service state and lifetime. The advantages of the encapsulation model is the natural object-oriented design that facilitates specialised extensions through *inheritance*. The main disadvantage is the poor fault tolerance due to the one-to-one association between the resource modelled and the Grid service as a single point of failure. The encapsulation approach has been taken by the currently obsolete Open Grid Service Infrastructure (OGSI) standard [163].
2. *Delegation* interposes a stateless Grid service between the client and the *driver* that manages the stateful resources (see Figure 2.6(b)). While the implementation of the Grid service is stateless, the *interface* of the service is *stateful*. The state of the service within the service interface is represented by the *context* [25] that identifies and maps a request to an existing stateful resource (for instance by providing its reference handler). The main advantage of the delegation model over encapsulation is the high



(a) Encapsulation (b) Delegation

Fig. 2.6. Stateful Grid service design alternatives.

degree of fault tolerance due to the m to n association between the stateless Grid service and the modelled resource (i.e., multiple Grid services can be used for accessing a stateful resource). The task of providing fault tolerance is naturally deferred to the specialised resource driver. The delegation approach has been taken by the Web Services Resource Framework (WSRF) [67] specification.

2.8 Grid Applications

This section presents three concrete Grid application models which are the subject of the performance, parameter, and optimisation analysis problems addressed by this thesis.

2.8.1 Single-Site Applications

The single-site Grid applications, where a Grid site has been defined in Definition 2.6, are typically represented by *sequential* and tightly coupled *parallel applications*. This section deals with the specification of the parallel application model that is the subject of the multi-experimental performance studies addressed by this thesis.

A parallel application consists of a set of distributed memory processes. Each process executes a program which is divided in sequential and parallel regions, as illustrated in Figure 2.7. A process may dynamically fork, synchronise, and terminate threads during its execution. All the threads of the process share the same address space. In a sequential region only one thread

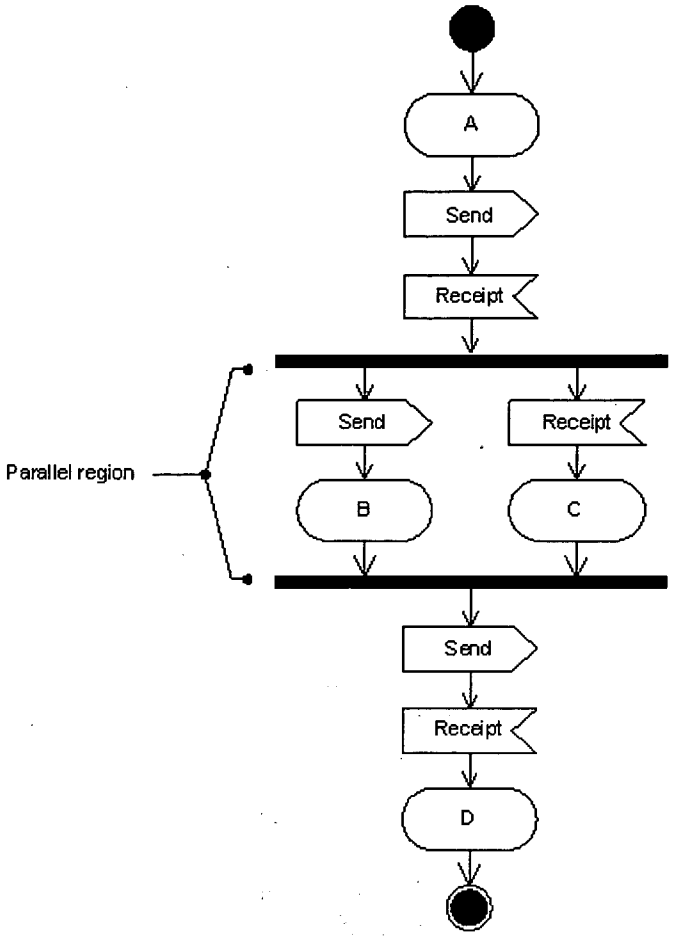


Fig. 2.7. The parallel application execution model.

of the process is active. In a parallel region several threads may be active and execute simultaneously. Depending on the language implementation, the threads may be spawned at the beginning of the program or at the beginning of each parallel region. At the end of the parallel region, the active threads may be synchronised, for instance through a barrier synchronisation or a join operation. Following the parallel region, all the parallel threads except the one that continues to execute the sequential region are either terminated or stopped. A stopped thread can be resumed by a subsequent parallel region or terminated at the end of the program execution. The threads active within the same process exchange data through a common shared memory. The distributed memory processes exchange data through generic `Send` and `Receive` message passing operations, executed either by the sequential processes or by

the parallel threads. All the parallel processes and all the threads are terminated at the end of the parallel application execution.

There are two emerged standards that implement in conjunction this hybrid distributed and shared-memory parallel application model:

1. *Message Passing Interface (MPI)* [145] for explicit message passing between processes on distributed memory architectures;
2. *Open Multi Processing (OpenMP)* [40] for implicit compiler-based parallelisation on shared memory architectures.

Figure 2.8 displays the typical scenario for executing parallel single-site applications on the Grid:

1. *query resource information* about the remote parallel computers and the underlying hardware and software configurations (e.g., CPU speed, memory size, disk size, compiler, software libraries) required to execute the application. Such a parallel computer represents a computational Grid site;
2. *transfer* the parallel application to the remote Grid site using the GridFTP file transfer protocol. Remotely running a (C or Fortran) parallel application is bound to difficult software dependencies, such as shared library availability, or non-standard compiler and link options. The easiest solution for solving such complex remote dependencies is to locally build *static binary executable code* compatible with (and potentially even optimised for) the remote architecture and operating system. In cases when the transfer of the source code to the remote Grid site cannot be avoided, the next execution steps are required;
3. *configure* the application source code for the target architecture, typically by forking a remote auto-configure program (e.g., GNU *Autoconf* [71]) using GRAM;
4. *build* (i.e., compile and link) the application, typically by forking a `make` command on the remote execution site front-end using GRAM;

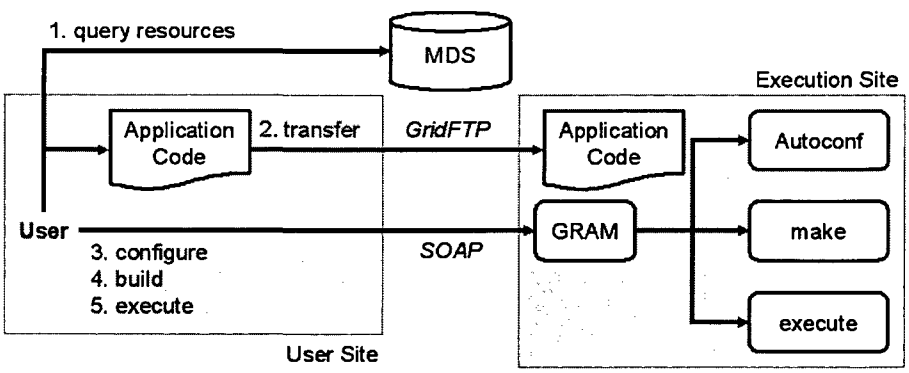


Fig. 2.8. Execution model of parallel applications on the Grid.

5. *execute* and *monitor* the application using GRAM, typically configured to interact with an available back-end job scheduler [108, 91, 179, 166, 152]. Automatic I/O file staging is automatically performed using the GASS functionality.

2.8.2 Workflow Applications

Workflow modelling is a well established area in computer science that has been strongly influenced by business process modelling work [173]. Recently, the Grid community has generally acknowledged that the orchestration of Grid services in a workflow represents an important class of loosely-coupled applications suited for programming large-scale Grid environments. The Grid services are usually wrappers around off-the-shelf applications (often also called components) that solve a well-defined atomic problem.

There is currently a large amount of research in the Grid community devoted to the specification of workflows application models, that range from low-level scripting languages [44, 101, 113, 154], to high-level abstract XML [92, 94, 167, 8, 49, 56, 89, 102], and user friendly graphical interfaces [19, 28, 52, 121]. The definition a new Grid workflow model is therefore beyond the tool development scope of this thesis. Rather, a low-level workflow model is adopted which is believed to constitute the minimal but sufficient foundation to which any higher-level workflow specification needs to be compiled.

Definition 2.7. *A* workflow application *is modelled by a Directed Graph (DG)* $\mathcal{A} = (Nodes, Edges)$*, where Nodes is the set of workflow* tasks *and Edges the set of* directed task dependencies*. Workflow tasks are classified into two distinct categories:* $Nodes = Nodes^{JS} \cup Nodes^{FT}$*:*

1. Job Submission, *denoted as* $JS(z) \in Nodes^{JS}$*, where z is the* abstract machine *where the JS task executes;*
2. File Transfer, *denoted as* $FT(z_1, z_2) \in Nodes^{FT}$*, where* z_1 *and* z_2 *are the source, respectively the destination abstract machines of the transfer.*

Let $succ(N)$ *denote the set of* successors *of one task* $N \in Nodes$*:*

$$N_s \in succ(N) \iff \exists\ (N, N_s) \in Edges.$$

Similarly, let $pred(N)$ *denote the set of* predecessors *of one task* $N \in Nodes$*:*

$$N_p \in pred(N) \iff \exists\ (N_p, N) \in Edges.$$

If $pred(N) = \phi$*, where* ϕ *denotes the empty set, then N is a* start task*. Similarly, if* $succ(N) = \phi$ *then N is an* end task*. Additionally, the set of predecessors and successors of rank p of a task N are referred as:*

$$pred^p(N) = pred(\ldots pred(N)),$$

respectively:

$$succ^p(N) = succ(\ldots succ(N))$$

(p calls). Two tasks N_1 *and* N_2 *are* independent *iff* $\nexists$ p *such that* $N_1 \in pred^p(N_2) \vee N_1 \in succ^p(N_2)$.

A *JS* task is modelled as a single-site Grid application, as described in Section 2.8.1, remotely allocated and manipulated using GRAM. A *FT* task uses the GridFTP high-performance network communication protocol to physically transfer a file between two (i.e., source and destination) Grid sites. GSI is employed for control flow task authentication as well as GridFTP control and data channel security.

A workflow can have an arbitrary number of start and end tasks. Workflow graph edges model pure control flow dependencies. In contrast to other traditional workflow approaches, the data communication is represented as separate *FT* workflow tasks and not as weights that annotate the graph edges. This representation looks more appropriate for data Grid workflows, where users often replicate data to locations with high-bandwidth access, without binding file transfers to immediate computation. Input and output *file staging* is modelled through *FT* workflow tasks having pre-defined fixed (instead of abstract) source, respectively destination machines.

2.8.3 Parameter Studies

Parameter studies, also known as parameter sweeps, are large sets of independent experiments that represent the same application executed on a different input parameter configuration. The scope of the parameter studies is to analyse the evolution of important output results as a function of various input parameter values.

Parameter studies can be modelled as a specialisation of the workflow model $\mathcal{A} = (Nodes, Edges)$ introduced in Definition 2.7, where:

1. the set of tasks exclusively consists of *JS* tasks: $Nodes = Nodes^{JS}$;
2. the set of *FT* (i.e., file transfer) tasks is empty: $Nodes^{FT} = \phi$;
3. the set of tasks dependencies is empty: $Edges = \phi$.

File staging is assumed to be performed off-line to the file systems of the Grid sites available to the parameter study.

3

The ZEN Experiment Specification Language

Existing parameter study tools provide support to specify value ranges for application parameters of interest, e.g., by means of external scripting languages [1], or through graphical annotation of input files [178]. All these approaches, however, force the user to export the application parameters to global input files or program arguments, which often requires undesired source code adaptation for using the tool.

Additionally, there are no tools that combine the experiment specification and management with cross-experiment performance analysis. All the currently existing performance tools are restricted to single experiment analysis, which is not enough for efficient application performance tuning, that is inherently a multi-experimental process.

Under this motivation, the ZEN language addresses the parameter specification problem for performance and parameter studies using a directive-based approach [123, 127]. So called *ZEN directives* are program comments that can be inserted in any source file to specify value ranges for arbitrary application parameters. The advantage of the directive-based approach over an external script is the ability to specify experiments at a more detailed granularity (e.g., associate local scopes to directives, restrict parametrisation to specific local variables, evaluate different scheduling alternatives for individual loops, or various distribution options for local parallel arrays). Moreover, the ZEN directives do not require source code modification and do not change the semantics of the code, as they are ignored by language processors that are unaware of their semantics. The ZEN directives are designed as language independent and therefore can be applied in the context of any programming language. Constraint directives are introduced to control and avoid meaningless experiments that could be generated by the cross product of the parameter sets defined. The scope of the ZEN language is not restricted to parameter studies. ZEN performance directives allow the user to specify a wide variety of performance metrics to be collected for arbitrary program regions.

3.1 ZEN Sets

An important goal in designing the ZEN language was to express wide value ranges for application parameters using a compact and practical syntax. For this purpose, this section introduces a special stand-alone language construct called ZEN set.

Definition 3.1. *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, defined by the Equation 3.1. An element of a ZEN set is called* ZEN element.

The ZEN sets have the following regular expression-based syntax:

zen-set	**is**	"{" *elem-list* "}"
elem-list	**is**	*elem* ["," *elem*]*
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*
integer	**is**	[+|−]?[0–9]
real	**is**	[+|−]?[0–9]+"."[0–9]*
zen-num-set	**is**	"{" *num-list* "}"
num-list	**is**	*num* ["," *num*]*
zen-string	**is**	([^\n{},:] | "\{" | "\}" | "\," | "\:")*

Let $\cdot$ denote the string concatenation operator, also referred in the following using one blank character. 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:

$$\varepsilon : \textit{zen-set} \rightarrow \mathcal{P}(\mathbb{R} \cup \textit{string}),\ \varepsilon\left(\bigcup_{i=1}^{n} \textit{elem}_i\right) = \bigcup_{i=1}^{n} \overline{\varepsilon}(\textit{elem}_i), \tag{3.1}$$

where *string* denotes an arbitrary string, $\mathcal{P}(\textit{string})$ denotes the set of strings, and the function $\overline{\varepsilon}$ is defined in Figure 3.1.

Informally, an *elem* construct of a ZEN set can be expressed as:

1. a regular real *number* (see Example 3.2, Equation 3.2);
2. a *low:up:stride* pattern evaluated to a sequence of numbers ranging from *low* to *up* with the increment *stride* (see Example 3.2, Equations 3.4 and 3.6). The stride is optional and has a default value of one, therefore:

$$\overline{\varepsilon} : elem \rightarrow \mathcal{P}(\mathbb{R} \cup string),$$

$$\overline{\varepsilon}(e) = \begin{cases} \{low + k * stride \mid k \in [0..\frac{up-low}{stride}]\}, & e \text{ is } low{:}up{:}stride; \\ \{e\}, & e \text{ is } number; \\ X, & e \text{ is } (zen\text{-}num\text{-}set \mid zen\text{-}string)+, \end{cases}$$

$$(zen\text{-}num\text{-}set \mid zen\text{-}string)+ = zen\text{-}string_1 \ \{num_{11}, \ldots, num_{1n_1}\} \ \ldots \\ zen\text{-}string_p \ \{num_{p1}, \ldots, num_{pn_p}\} \ zen\text{-}string_{p+1},$$

$$X = \{\overline{\varepsilon}_s(zen\text{-}string_1) \ n_1 \ldots \overline{\varepsilon}_s(zen\text{-}string_p) \ n_p \ \overline{\varepsilon}_s(zen\text{-}string_{p+1}) \mid \\ \forall (n_1, \ldots, n_p) \in \varepsilon(\{num_{11}, \ldots, num_{1n_1}\}) \times \ldots \times \varepsilon(\{num_{p1}, \ldots, num_{pn_p}\})\},$$

$$\overline{\varepsilon}_s : string \rightarrow string, \ \overline{\varepsilon}_s(s) = \begin{cases} s, & \forall e \in \{\text{“\textbackslash”}, \text{“,”}, \text{“\textbackslash\{”}, \text{“\textbackslash\}”}, \text{“\textbackslash :”}\}, \ e \notin s; \\ \overline{\varepsilon}_s(s_l) \ c \ \overline{\varepsilon}_s(s_r), & s = s_l \setminus c \ s_r, \ \forall c \in \{\text{‘,’}, \text{‘\{’}, \text{‘\}’}, \text{‘ :’}\}. \end{cases}$$

Fig. 3.1. The ZEN set element evaluation function.

$$\overline{\varepsilon}(low{:}up) = \overline{\varepsilon}(low{:}up{:}1);$$

3. a composite element *comp-elem* that alternates multiple sets of numbers denoted as *zen-num-set*, with multiple ZEN strings denoted as *zen-string*. The composite element is evaluated to a set of ZEN elements by computing the cross product of the *zen-num-set*s and replacing each *zen-num-set* with the corresponding tuple element (see Example 3.2, Equations 3.7, 3.9, 3.11, and 3.13). The *zen-string* elements must obey the syntax defined by the evaluation function $\overline{\varepsilon}_s$ in Figure 3.1, which can be informally translated to the following rules:
   a) *commas* inside a *zen-string* must be prefixed by one '\' character which distinguishes them from the value delimiters of a *zen-num-set*:

   $$\overline{\varepsilon}(string_l \setminus, \ string_r) = string_l \ , \ string_r,$$

   where ‘,’ $\notin string_l \ \wedge$ ‘,’ $\notin string_r$ (see Example 3.2, Equations 3.8, 3.10, 3.11, 3.12, and 3.13);
   b) *braces* inside a *zen-string* must be prefixed by one '\' character which avoids *zen-num-set*s inside *zen-string*s:

   $$\overline{\varepsilon}(\setminus\{num_1\setminus, \ldots \setminus, num_n\setminus\}) = \{num_1, \ldots, num_n\}$$

   (see Example 3.2, Equations 3.8 and 3.10);
   c) *colons* inside a *zen-string* must be prefixed by one '\' character which allows the pattern *low:up:stride* be a *zen-string*:

   $$\overline{\varepsilon}(low\setminus : up\setminus : stride) = low : up : stride$$

   (see Example 3.2, Equations 3.5, 3.10, and 3.12).

Example 3.2 (ZEN set evaluation examples).

- Numerical value set enumeration:

$$\varepsilon(\{1,2,3\}) = \{1,2,3\}; \tag{3.2}$$

- Alphanumerical (i.e., ZEN string) value set enumeration:

$$\varepsilon(\{a,b,c\}) = \{a,b,c\}; \tag{3.3}$$

- Numerical value ranges using the *low:up:stride* pattern:

$$\varepsilon(\{1:10:2\}) = \{1,3,5,7,9\}; \tag{3.4}$$

- The *low:up:stride* pattern as ZEN string through colon escape:

$$\varepsilon(\{1\backslash:10\backslash:2\}) = \{1:10:2\}; \tag{3.5}$$

- Mixed numerical enumeration and *low:up:stride* value range:

$$\varepsilon(\{0,1:10:2,11\}) = \{0,1,3,5,7,9,11\}; \tag{3.6}$$

- Function parameter variation:

$$\varepsilon(\{foo(\{10,20,30\})\}) = \{foo(10), foo(20), foo(30)\}; \tag{3.7}$$

- Inner *zen-num-set* avoidance through brace, and comma escape:

$$\varepsilon(\{foo(\backslash\{10\backslash,20\backslash,30\backslash\})\}) = \{foo(\{10,20,30\})\}; \tag{3.8}$$

- Array distribution variation [88]:

$$\varepsilon(\{\texttt{BLOCK}(\{4:12:4\}),\texttt{CYCLIC}(\{8,16\})\}) = \{\texttt{BLOCK}(4),\texttt{BLOCK}(8),\texttt{BLOCK}(12),\texttt{CYCLIC}(8),\texttt{CYCLIC}(16)\}; \tag{3.9}$$

- Inner *zen-num-set* avoidance through brace, colon, and comma escape:

$$\varepsilon(\{\texttt{BLOCK}(\backslash\{4\backslash:12\backslash:4\backslash\})\backslash,\texttt{CYCLIC}(\backslash\{8\backslash,16\backslash\})\}) = \{\texttt{BLOCK}(\{4:12:4\}),\texttt{CYCLIC}(\{8,16\})\}; \tag{3.10}$$

- Two-dimensional matrix index annotation through comma escape:

$$\varepsilon(\{A(\{0:10:5\}\backslash,\{4:12:4\})\}) = \{A(0,4),A(0,8),A(0,12),A(5,4),A(5,8),A(5,12),A(10,4),A(10,8),A(10,12)\}; \tag{3.11}$$

- One-dimensional matrix index annotation through comma and colon escape:

$$\varepsilon(\{A(\{0:10:5\}\backslash,4\backslash:12\backslash:4)\}) = \{A(0,4:12:4),A(5,4:12:4),A(10,4:12:4)\}; \tag{3.12}$$

- Loop scheduling variation [40][1]:

$$\varepsilon(\{\texttt{STATIC}\backslash, \{4,8\}, \texttt{DYNAMIC}\backslash, \{1:4\}\}) = \{\underline{\texttt{STATIC},4}\ ,\ \underline{\texttt{STATIC},8}\ ,\ \underline{\texttt{DYNAMIC},1}\ ,\ \underline{\texttt{DYNAMIC},2}\ ,\underline{\texttt{DYNAMIC},3}\ ,\ \underline{\texttt{DYNAMIC},4}\}. \quad (3.13)$$

The *total order* of the ZEN elements, denoted by the operator $\prec$, in a ZEN set $\textit{zen-set} = \bigcup_{i=1}^{n} \textit{elem}_i$ is given by the following ordering rules:

1. The order of the comma-separated elements is the enumeration order (see Example 3.2, Equations 3.2, 3.3, 3.7, 3.10, and 3.13):

$$\forall\ \textit{elem}_i, \textit{elem}_j \in \varepsilon\left(\bigcup_{i=2}^{n} \textit{elem}_i\right),\ \forall\ i,j \in [1..n],\ \textit{elem}_i \prec \textit{elem}_j \iff i < j;$$

2. The element order specified by a *low:up:stride* value range pattern is the element sequence from *low* to *up* with the increment *stride* (see Example 3.2, Equations 3.4, 3.6, 3.12, and 3.13):

$$\forall\ e_i, e_j \in \overline{\varepsilon}(\textit{low:up:stride}),\ e_i \prec e_j \iff$$
$$e_i = \textit{low} + k_i * \textit{stride}\ \wedge\ e_j = \textit{low} + k_j * \textit{stride}\ \wedge\ k_i < k_j;$$

3. The cross product tuples are ordered lexicographically (see Example 3.2, Equations 3.9, 3.11, and 3.13):

$$\forall\ (n_1,\ldots,n_p),(n'_1,\ldots,n'_p) \in \varepsilon\left(\bigcup_{i=1}^{n_1} \textit{num}_{1i}\right) \times \ldots \times \varepsilon\left(\bigcup_{j=1}^{n_p} \textit{num}_{pj}\right),$$
$$\textit{string}_1\ n_1 \ldots \textit{string}_p\ n_p\ \textit{string}_{p+1} \prec \textit{string}_1\ n'_1 \ldots \textit{string}_p\ n'_p\ \textit{string}_{p+1}$$
$$\iff \exists\ i \in [1..n]\ \textit{such that}\ \left(\forall\ j \in [1..i-1] : n_j = n'_j\right)\ \wedge\ n_i \prec n'_i.$$

Definition 3.3. *Let $(A, \prec)$ and $(B, \prec)$ denote two totally ordered sets with the same ordering operation $\prec$. The* union *of the totally ordered sets A and B is the totally ordered set $(A \cup B, \prec)$ obtained by appending $B\backslash A$ to A:*

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

The total order of ZEN sets is used by the ZEN index constraint directive, which will be introduced in Section 3.7.

[1] To avoid any potential confusion and allow the reader distinguish between commas as set element delimiters and commas as regular characters of a string, the ZEN elements have been underlined.

3.2 ZEN Directives

Definition 3.4. *A* ZEN directive *is a comment line that starts with the prefix ZEN$.*

The characters that mark the beginning (and eventually the end) of a comment are the only programming language specific features of ZEN. Example 3.5 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 3.5 (Sample ZEN directives in various programming languages).

```
!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 }
```

The ZEN language defines four categories of directives:

1. *Substitute directives* (see Section 3.5) and
2. *Assignment directives* (see Section 3.4) assign a ZEN set to an application parameter. Each ZEN element of the ZEN set represents an experimental value for the corresponding parameter;
3. *Constraint directives* (see Section 3.7) define boolean conditions over multiple ZEN variables which restricts the set of possible experiments to a meaningful subset;
4. *Performance directives* (see Section 3.8) are used to request a wide variety 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 to which the directive is applied.

Definition 3.6. *A* ZEN variable *is an arbitrary application parameter defined by a ZEN substitute or a ZEN assignment directive. A ZEN variable is a sequence of characters that must obey the following syntax constraints:*

1. *equality and blank characters must be prefixed by a '\' character, which distinguishes them from the assignment character and eventual neighbouring blank characters in a ZEN directive (e.g.,* `count\=4` *in Example 7.2);*
2. *arithmetical (+, −, *, /, %, ^), relational (==, !=, <, >, <=, >=), and logical(!, &&,||) operators, as well as left and right parentheses must be prefixed by a '\' character, which distinguishes them from the parentheses and the operators of a ZEN constraint; (e.g.,* `BLOCK\(4\)` *in Example 3.11);*

In the following, several definitions which express the semantics of the ZEN directives are presented.

Definition 3.7. *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:*

$$\mathcal{V}^z = \varepsilon(\mathcal{S}),$$

where the value function ε and the operator $\prec$ have been defined in Section 3.1.

The need for the total order of the value set will be addressed in Section 3.7.

Definition 3.8. *An arbitrary file $\mathcal{Z}$ (e.g., source, input data, makefile) augmented with a set of ZEN directives is called* ZEN file, *denoted as $\mathcal{Z}(z_1, \ldots, z_n)$, where z_i are the ZEN variables defined by the ZEN directives of $\mathcal{Z}$, $\forall\, i \in [1..n]$. A* ZEN file instance *denoted as $\mathcal{ZI}(e_1, \ldots, e_n)$, where $e_i \in \mathcal{V}^{z_i}$, $\forall\, i \in [1..n]$, is an instantiation of the ZEN file $\mathcal{Z}$, obtained by instantiating each ZEN variable with one ZEN element from its value set.*

Informally, a ZEN file represents a parameterised application file. A ZEN file instance instantiates each application parameter of the ZEN file with one concrete parameter value. For instance, Example 3.10 illustrates an excerpt of a ZEN file that is denoted as $\mathcal{Z}$(`NUM_THREADS(4)`).

In the cases when the ZEN variables are irrelevant, the ZEN files and the ZEN file instances will be simply denoted as $\mathcal{Z}$, respectively $\mathcal{ZI}$.

3.3 ZEN Transformation System

The generation of the ZEN file instances described by a ZEN file is performed by the *ZEN Transformation System* depicted in Figure 3.2. The ZEN Transformation System can be seen as a source-to-source language processor. The scanner and parser modules examine the ZEN directives and construct an abstract syntax tree representation of the ZEN file. The code generator is different from a conventional compiler unparser, as it commonly generates a possibly large number of ZEN file instances. The code generation rules are specified by the semantics of the ZEN directives that annotate the ZEN file. The number of the ZEN file instances is given by the cardinality of the value set of the ZEN file which will be defined in Section 3.6.

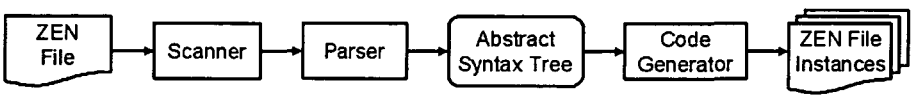


Fig. 3.2. The ZEN Transformation System.

The ZEN variables can be of three different types: *integer*, *real* and *string*. Introducing the integer and real types along side string (which otherwise would have sufficed) is motivated by the value set constraints which will be described Section 3.7.

Definition 3.9. *The* type τ *of a ZEN variable* z *is determined by the ZEN Transformation System in the parsing phase based on the values of the associated ZEN elements, as follows:*

$$\tau(z) = \begin{cases} \text{“integer”}, \forall\ e \in \mathcal{V}^z,\ e \text{ is integer}; \\ \text{“real”}, \quad \forall\ e \in \mathcal{V}^z,\ e \text{ is number} \wedge \neg\ (\forall\ e \in \mathcal{V}^z,\ e \text{ is integer}); \\ \text{“string”}, \forall\ e \in \mathcal{V}^z,\ e \text{ is zen-string} \wedge \neg\ (\forall\ e \in \mathcal{V}^z,\ e \text{ is number}). \end{cases}$$

3.4 ZEN Substitute Directive

The *ZEN substitute directive* employs a conventional macroprocessor-based string replacement mechanism to overwrite application parameters with value instances of interest within ZEN files. This is expressed by assigning a ZEN set to a ZEN variable. This directive is commonly employed to examine various language-specific parallelisation patterns, like e.g., problem and machine sizes, data distributions, or work scheduling strategies. The scope of the *global substitute directive* comprises the entire ZEN file where the directive is defined. The global substitute directive has the following syntax:

substitute-directive is SUBSTITUTE *zen-var* = *zen-set*
zen-var is ([^-\+*/%"^"=<>!&\|\(\): \t\r\n\f]|"\="|
"\+"|"\-"|"*"|"\/"|"\%"|"\^"|"\==”|"\!="|
"\ <"|"\ >"|"\ <="|"\ >="|"\!"|"\&&"|"\||"|
"\("|"\)"|"\:"|"\")+

The ZEN Transformation System replaces all the occurrences in the entire file of the name of a ZEN variable z with one element $e \in \mathcal{V}^z$. It is the task of the user to verify that the global substitution produces a correct outcome. Eventual erroneous substitutions usually produce subsequent faulty file compilations or faulty application executions.

Example 3.10 (OpenMP parallel region).

```
!ZEN$ SUBSTITUTE NUM_THREADS\(4\) = { NUM_THREADS({1:4}) }
!$OMP PARALLEL NUM_THREADS(4)
. . .
!$OMP END PARALLEL
```

OpenMP [40] is a directive-based language which represents the de-facto standard for programming shared memory architectures (see Section 2.8.1).

One typical optimisation problem for OpenMP applications is to determine the optimal number of threads which execute a parallel region, expressed by the NUM_THREADS clause of the PARALLEL directive. The global ZEN substitute directive illustrated in Example 3.10 substitutes the string NUM_THREADS(4) with the ZEN elements from the set:

$$\mathcal{V}^{\texttt{NUM\_THREADS(4)}} = \{\texttt{NUM\_THREADS(i)} \mid \forall\, i \in \{1,2,3,4\}\}.$$

As a result, four ZEN file instances are generated, each one executing the loop using a different number of parallel threads. Note that the code shown in this example is semantically valid for both ZEN-aware and ZEN-unaware compilers (i.e., that understand or ignore the ZEN directives). The OpenMP parallel regions generated within each ZEN file instance are depicted in Figure 3.3.

File Instance ($\mathcal{ZI}$)	Generated Code
$\mathcal{ZI}$(NUM_THREADS(1))	!$OMP PARALLEL NUM_THREADS(1)
$\mathcal{ZI}$(NUM_THREADS(2))	!$OMP PARALLEL NUM_THREADS(2)
$\mathcal{ZI}$(NUM_THREADS(3))	!$OMP PARALLEL NUM_THREADS(3)
$\mathcal{ZI}$(NUM_THREADS(4))	!$OMP PARALLEL NUM_THREADS(4)

Fig. 3.3. The file instances generated by Example 3.10.

3.4.1 Local Substitute Directive

It often occurs in practice that the user needs to apply a parameter substitution to a specific restricted code region, for instance to a certain OpenMP loop from a file that contains many other loops. The local ZEN substitute directive restricts the scope of the global version to a specific region of the ZEN file through the following syntax:

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

The local substitute directives can be nested.

High Performance Fortran (HPF) [88] is a directive-based language designed in the late 1990s to improve the productivity of writing data parallel programs. Despite failing the general acceptance in the scientific community due to the lack of performance delivered compared to MPI, HPF deserves further attention as a high-productivity paradigm for programming next generation computing architectures [147].

To examine the scalability of HPF programs, the user commonly varies the number of parallel processors expressed through a PROCESSORS directive. The HPF code shown in Example 3.11 defines an 8 × 8 two-dimensional processor

array. The local ZEN substitute directive $d1$ causes the replacement of all the occurrences of the string *"P(8,8)"* with every element in the associated value set:

$$\mathcal{V}^{P(8,8)} = \{P(8,2), P(8,4), P(10,2), P(10,4), P(12,2), P(12,4), P(14,2), P(14,4)\}.$$

Example 3.11 (HPF array and independent loop distributions).

```
d1: !ZEN$ SUBSTITUTE P\(8,8\) = { P({8:15:2}, {2,4}) } BEGIN
    !HPF$ PROCESSORS P(8,8)
d1: !ZEN$ END SUBSTITUTE
    . . .
d2: !ZEN$ SUBSTITUTE BLOCK\(4\) = { BLOCK({4:10:2}),
                                    CYCLIC({10,20}) } BEGIN
    !HPF$ DISTRIBUTE A(BLOCK(4)) ONTO P
d2: !ZEN$ END SUBSTITUTE
    . . .
d3: !ZEN$ SUBSTITUTE A\(i\) = { A(i), B(I(i)) } BEGIN
    !HPF$ INDEPENDENT, ON HOME(A(i))
d3: !ZEN$ END SUBSTITUTE
    DO i = 1, N
      . . . A(i) . . .
      . . . B(I(i)) . . .
    ENDDO
```

Beyond the specification of appropriate machine sizes, the array distribution is another non-trivial optimisation that can significantly influence the overall performance of the parallel HPF applications. The local ZEN substitute directive $d2$ in Example 3.11 defines a ZEN variable `BLOCK(4)` with the value set:

$$\mathcal{V}^{\texttt{BLOCK(4)}} = \{\texttt{BLOCK(4)}, \texttt{BLOCK(6)}, \texttt{BLOCK(8)}, \texttt{BLOCK(10)}, \texttt{CYCLIC(10)}, \texttt{CYCLIC(20)}\}.$$

Every ZEN element represents a potentially good array distribution that substitutes the original `BLOCK(4)` distribution.

The ZEN substitute directive can be similarly employed to examine different options of the HPF `REDISTRIBUTE` directive.

The `HPF ON` and the `ON HOME` directives allow the programmer to control the distribution of the computation across the processors of a parallel machine. The `ON HOME` directive requests the work distribution of a parallel loop be derived according to an array section provided as argument. Such loops often contain references to array elements that are distributed using various irregular patterns for which is hard to figure out the optimal distribution of

iterations. Example 3.11 defines an HPF `INDEPENDENT` loop which accesses the elements of two arrays A and B. The local ZEN substitute directive $d3$ specifies two different schedules for the loop iteration i: the processor $A(i)$ and the processor $B(I(i))$. The local substitute directive insures that the string `A(i)` is replaced only in the `INDEPENDENT` directive and not further in the parallel loop.

3.4.2 Homonym ZEN Variables

Appropriate scheduling of parallel loops is another critical optimisation decision for OpenMP parallel programs. Example 3.12 contains two OpenMP parallel loops for which various scheduling strategies are examined by means of ZEN directives.

Example 3.12 (OpenMP loop scheduling).

```
d1: !ZEN$ SUBSTITUTE STATIC = { STATIC\,{1,10:100:10},
                                DYNAMIC\,{1,10:100:10} }
    !$OMP PARALLEL DO SCHEDULE(STATIC) NUM_THREADS(4)
    . . .
d2: !ZEN$ SUBSTITUTE STATIC = { GUIDED } BEGIN
d3: !$OMP PARALLEL DO SCHEDULE(STATIC) NUM_THREADS(4)
d2: !ZEN$ END SUBSTITUTE
```

The global ZEN substitute directives $d1$ examines `STATIC` and `DYNAMIC` scheduling strategies combined with different chunk sizes for all the parallel loops of the ZEN file. `STATIC` scheduling means that the iterations are assigned to all the parallel threads (i.e., four in this example) statically, before the parallel loop starts its execution. `DYNAMIC` scheduling means that each thread dynamically receives a new set of iterations after it finishes the iterations assigned [40]. The chunk size indicates the number of loop iterations to be scheduled atomically. The directive $d1$ replaces the original OpenMP scheduling clause `STATIC` with every ZEN element $e \in \mathcal{V}^{\texttt{STATIC}}$ in different ZEN file instances.

The local ZEN directive $d2$ has the scope restricted to the parallel loop directive $d3$. The `GUIDED` scheduling means that the iteration space is divided into scheduling pieces, where the size of each successive piece is exponentially decreased [40].

One can notice in this example that the ZEN directives $d1$ and $d2$ define two ZEN variables that have identical name `STATIC`. Despite their identical name, the two ZEN variables are distinct, each one having its own scope and value set. Intentionally or not, such situations often happen in practice and need special care which is the subject of this section. In this example, keeping the default `STATIC` distribution for both parallel loops, as also a semantically proper ZEN variable naming (i.e., `STATIC`), may be of importance for the user.

Definition 3.13. *If the textual name of two or more ZEN variables in a ZEN file is identical, these ZEN variables are called* homonyms.

The impact of the homonym ZEN variables on the semantics of the global and local ZEN substitute directives is as follows:

1. No homonym global ZEN substitute variables are allowed within one ZEN file;
2. A local ZEN substitute directive d_i with a ZEN variable z_i, defined in the scope of any global or local ZEN substitute directive d_j with an associated ZEN variable z_j, where z_i and z_j are homonym (i.e., $\nu(z_i) = \nu(z_j)$) augments the value set of z_j as follows:

$$\mathcal{V}^{z_j} = \mathcal{V}^{z_j} \cup \mathcal{V}^{z_i},$$

where the union of two totally ordered value sets has been defined in Definition 3.3.

A ZEN variable z is therefore characterised by the:

1. *textual name* denoted in the following as $\nu(z)$;
2. *ZEN directive* d which assigns a value set $\mathcal{V}^z$ to z;
3. *ZEN file* $\mathcal{Z}$ which contains the directive d.

The following conventions for naming ZEN variables hold for the remainder of the thesis:

1. if no homonym ZEN variable has been defined, the plain textual name of the ZEN variable is used;
2. if other homonym ZEN variables have been defined, the ZEN variable is referred through its textual name subscripted with a unique ZEN directive identifier.

Therefore, the directive $d2$ from Example 3.12 defines the following value set for the ZEN variable $\texttt{STATIC}_{d2}$:

$$\mathcal{V}^{\texttt{STATIC}_{d2}} = \mathcal{V}^{\texttt{STATIC}_{d1}} \cup \{\texttt{GUIDED}\}.$$

The substitute directive must be used with care, as it might replace undesired occurrences of the ZEN variable in the corresponding scope. For instance, if the variable D in Example 3.14 must be substituted in a given scope, then every occurrence of this character would be replaced, even in keywords such as `DO` or `END`. This problem is particularly critical for shortly named variables (e.g., one character long) that are commonly used by the programmers (even as global external variables), which are problematic or simply inconvenient to be renamed. To overcome this limitation and give the user extra flexibility, the ZEN assignment directive is introduced.

3.5 ZEN Assignment Directive

The *ZEN assignment directive* is used to insert assignment statements into ZEN files. Its purpose is to indicate all the values of interest for a specific program variable, which must be defined in the context of the directive location in the ZEN file. Formally, a ZEN assignment directive assigns a ZEN set to a ZEN variable using the following syntax (where *zen-var* has been defined in Section 3.4 and *zen-set* in Section 3.1):

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

The ZEN Transformation System, introduced in Section 3.3, textually replaces a ZEN assignment directive with a statement which assigns one element $e \in \mathcal{V}^z$ to the ZEN variable z. The assignment statement must conform to the syntax of programming language in which the ZEN file is written. For example, if the ZEN file represents a C program, the assignment statement must adhere to the C language syntax. The ZEN Transformation System does not apply any type checking or examine whether the (ZEN) variable has been declared in the scope of the directive. An eventual *"variable not found"* syntax error will be detected by a subsequent compilation of the ZEN file instance.

Example 3.14 (Shortly named ZEN variables).

```
      INTEGER D, i
s:    D = 50
d:    !ZEN$ ASSIGN D = { 2**{6:12} }
      DO i = 1, D
```

The ZEN assignment directive d in Example 3.14 assigns seven values to a ZEN variable D that represents the upper bounds of the immediately following `DO` loop:

$$\mathcal{V}^D = \{2**6, 2**7, 2**8, 2**9, 2**10, 2**11, 2**12\}.$$

Note that the code is semantically valid for both ZEN-aware and ZEN-unaware compilers. The ZEN-aware compilers replace the ZEN directive with an assignment statement that assigns one element $e \in \mathcal{V}^D$ to the (ZEN) variable D. In this example the default assignment s becomes redundant and is subject for compiler dead-code elimination. Also note that using a substitution in place of the assignment directive would also replace the character `D` in the keyword `DO` which would produce an erroneous program.

3.6 Multi-Dimensional Value Set

It is clear that one ZEN directive implies a number of ZEN file instances equal to the cardinality of the value set defined. This section describes how

multiple ZEN directives defined within a single ZEN file impact on the ZEN file instances generated.

Definition 3.15. *The* multi-dimensional value set *of n distinct ZEN variables* $z_1, \ldots, z_n$*, denoted as* $\mathcal{V}(z_1, \ldots, z_n)$*, is the cross 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* $\mathcal{Z}(z_1, \ldots, z_n)$*, denoted by* $\mathcal{V}(\mathcal{Z}(z_1, \ldots, z_n))$*, or simply by* $\mathcal{V}^{\mathcal{Z}}$*, is the entire set of ZEN file instances generated from the multi-dimensional value set of its ZEN variables:*

$$\mathcal{V}(\mathcal{Z}(z_1, \ldots, z_n)) = \{\mathcal{ZI}(e_1, \ldots, e_n) \mid \forall\ (e_1, \ldots, e_n) \in \mathcal{V}(z_1, \ldots, z_n)\}.$$

For instance, Example 3.12 defines two ZEN directives $d1$ and $d2$, whose multi-dimensional value set is given by the cross product of their value sets:

$$\mathcal{V}(\texttt{STATIC}_{d1}, \texttt{STATIC}_{d2}) = \mathcal{V}^{\texttt{STATIC}_{d1}} \times \mathcal{V}^{\texttt{STATIC}_{d2}},$$

with the cardinality:

$$|\mathcal{V}(\texttt{STATIC}_{d1}, \texttt{STATIC}_{d2})| = \left|\mathcal{V}^{\texttt{STATIC}_{d1}}\right| \times \left|\mathcal{V}^{\texttt{STATIC}_{d2}}\right| = 22 \times 23 = 506.$$

Definition 3.16. *A* ZEN application, *denoted by* $\mathcal{A}(\mathcal{Z}_1, \ldots, \mathcal{Z}_n)$*, or simply by* $\mathcal{A}$*, consists of a set of ZEN files* $\mathcal{Z}_1, \ldots, \mathcal{Z}_n$*. A* ZEN application instance, *denoted by* $\mathcal{AI}(\mathcal{ZI}_1, \ldots, \mathcal{ZI}_n)$*, or simply by* $\mathcal{AI}$*, is a set of ZEN file instances which instantiate each ZEN file of the ZEN application:*

$$\mathcal{AI}(\mathcal{ZI}_1, \ldots, \mathcal{ZI}_n) = \left\{\mathcal{ZI}_i \in \mathcal{V}^{\mathcal{Z}_i} \mid \forall\ i \in [1..n]\right\}.$$

From an informal perspective, a ZEN application represents a Grid application annotated with ZEN directives that confirms to one of the models presented in Section 2.8.

Definition 3.17. *The* value set *of a ZEN application, denoted in the following by* $\mathcal{V}(\mathcal{A}(\mathcal{Z}_1, \ldots, \mathcal{Z}_n))$ *or simply by* $\mathcal{V}^{\mathcal{A}}$*, is the set of application instances generated by the cross product of the value sets of its constituent ZEN files:*

$$\mathcal{V}(\mathcal{A}(\mathcal{Z}_1, \ldots, \mathcal{Z}_n)) = \{\mathcal{AI}(\mathcal{ZI}_1, \ldots, \mathcal{ZI}_n) \mid \forall\ (\mathcal{ZI}_1, \ldots, \mathcal{ZI}_n) \in \mathcal{V}^{\mathcal{Z}_1} \times \ldots \times \mathcal{V}^{\mathcal{Z}_n}\}.$$

3.7 ZEN Constraint Directive

The plain cross product of the value sets often produces a large number of ZEN element combinations that have no useful practical meaning. The consequence can be a dramatic increase in the number of experiments and the time needed

to conduct them, for instance in the context of a parameter study. The *ZEN constraint directive* is introduced with the purpose of filtering the meaningless or irrelevant the parameter combinations from the multi-dimensional value set.

Similarly to the substitute directive, the ZEN constraint directives can have global and local scopes. The local ZEN constraint directives can also be nested. The syntax of the ZEN constraint directive is as follows:

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 which contains 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: $\{!, \&\&, ||\}$. The symbols % and $\hat{}$ denote the modulo, respectively the power operators. The operators assume the standard mathematical associativity which can be overwritten by using parentheses. The arithmetical operators have precedence over the relational operators, which have precedence over the logical operators. An arithmetical operation over a set of integers produces an integer result. An operation over a set of mixed integer and real numbers produces a real result.

There are two types of ZEN variables that can appear in a ZEN constraint:

1. *local ZEN variables* that must be defined in the scope of the ZEN constraint;
2. *external ZEN variables* that must be globally defined in a different ZEN file, referred by prefixing the ZEN variable with the ZEN file name followed by a colon (see Example 7.15).

A ZEN constraint directive denoted as 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*$_n$ }. If there exist homonym ZEN variables in the scope of the directive with the name in { *zen-var*$_1$, ..., *zen-var*$_n$ }, the following set of constraints is generated:

$$\{bool\text{-}expr(z_1, \ldots, z_n) \mid \forall \{z_1, \ldots, z_n\} \subset scope(d), \\ such\ that\ \nu(z_i) = zen\text{-}var_i,\ \forall\, i \in [1..n]\},$$

where $\nu(z_i)$ is the textual name of a ZEN variable, as defined in Section 3.4.2.

The ZEN constraint directive defines two types of constraints, which depend on the type of the ZEN variables involved (see Definition 3.9):

1. *value set constraint* defines a boolean expression over a set of ZEN variables of type integer and real (see Section 3.7.1);
2. *index domain constraint* defines a boolean expression over a set of ZEN variables of any type, including string (see Section 3.7.2).

3.7.1 Value Set Constraint

The *value set constraint*, indicated by the `VALUE` clause of the ZEN constraint directive, defines a boolean expression over a set of ZEN variables of types integer and real. The type of a ZEN variable has been defined in Definition 3.9.

Definition 3.18. *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 called* value-valid *iff the following condition holds:*

$$valid(e_1, \ldots, e_n) \iff \alpha(\Pi_{j_1,\ldots,j_m}(e_1, \ldots, e_n)) = true,$$

$\forall\ \alpha : \mathcal{V}^{z_{j_1}} \times \ldots \times \mathcal{V}^{z_{j_m}} \rightarrow$ *boolean a value set constraint, where:*

$$\{z_{j_1}, \ldots, z_{j_m}\} \subset \{z_1, \ldots, z_n\},\ \forall\ j_k \in [1..n],\ \forall\ k \in [1..m]\ \wedge\ m < n.$$

The notation $\Pi_{j_1,\ldots,j_m}(e_1, \ldots, e_n)$ *denotes the* projection *of the tuple element* $(e_1, \ldots, e_n)$ *from the n-dimensional space* $\mathcal{V}^{z_1} \times \ldots \times \mathcal{V}^{z_n}$ *onto its m-dimensional subspace* $\mathcal{V}^{z_{j_1}} \times \ldots \times \mathcal{V}^{z_{j_m}}$.

Informally, a tuple $(e_1, \ldots, e_n) \in \mathcal{V}^{z_1} \times \ldots \times \mathcal{V}^{z_n}$ is value-valid iff it satisfies all the value set constraints defined across any subset of the ZEN variables involved. A value set constraint is evaluated by instantiating each ZEN variable z_i with the corresponding ZEN element e_i from the tuple, $\forall\ i \in [1..n]$. All the invalid tuples are eliminated from the multi-dimensional value set.

Example 3.19 (Value set constraint).

```
INTEGER D, P, i
!ZEN$ ASSIGN P = { {8:16:4}**2 }
D = 50
!ZEN$ ASSIGN D = { 2**{6:12} }
DO i = 1, D
!ZEN$ CONSTRAINT VALUE D^3 / P < 40000000
```

In Example 3.19, the ZEN variable D defines the powers of 2 from 2^6 to 2^{12}. The ZEN variable P defines the square numbers from 2^2 to 8^2 with the stride 2:

$$\mathcal{V}^D = \{2**6, 2**7, 2**8, 2**9, 2**10, 2**11, 2**12\};$$
$$\mathcal{V}^P = \{8**2, 12**2, 16**2\}.$$

The value set constraint directive filters the ZEN elements from the cross product $\mathcal{V}^N \times \mathcal{V}^P$, such that the boolean expression defined yields true (see Figure 3.4):

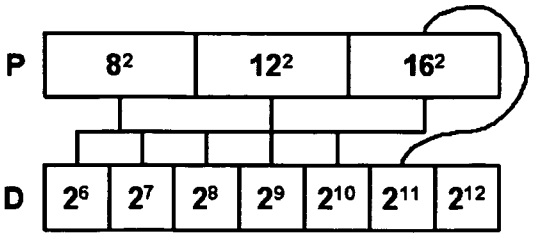


Fig. 3.4. The value set constraint defined by the Example 3.19.

$$\mathcal{V}(D,P) = \left\{ (e_1, e_2) \mid \frac{e_1^3}{e_2} < 40000000,\ \forall\ e_1 \in \mathcal{V}^N,\ \forall\ e_2 \in \mathcal{V}^P \right\}.$$

Assuming that D represents the size of a three-dimensional array and P the number of the available processors onto which the array is distributed, the constraint restricts the value set to those combinations which need less than $40MB$ on each processor.

3.7.2 Index Domain Constraints

While the value set constraint boolean expression is meaningful to be defined over a set of ZEN variables of types integer and real, it is problematic to comprise ZEN variables of type string. For this reason, the value set of a ZEN variable has been defined as a totally ordered set (see Definition 3.7) that associates a well-defined index to each ZEN element, as specified by the following definition.

Definition 3.20. *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 set of positive natural numbers (i.e., non-zero). The total order of elements in $\mathcal{I}^z$ is the natural element order. The value function *of a ZEN variable z is the total bijective function:*

$$\vartheta : \mathcal{I}^z \rightarrow \mathcal{V}^z,$$

which associates each element $\vartheta(i) \in \mathcal{V}^z$ with an index $i \in \mathcal{I}^z$ *such that:*

$$\forall\ i, i_1, i_2 \in \mathcal{I}^z,\ i_1 < i < i_2 \iff \vartheta(i_1) \prec \vartheta(i) \prec \vartheta(i_2).$$

The index function*:*

$$\vartheta^{-1} : \mathcal{V}^z \rightarrow \mathcal{I}^z$$

is the inverse of the value function.

The ZEN directives defined by Example 3.19 shown in the previous section define the following index sets and value functions:

$$\mathcal{I}^D = \{1,2,3,4,5,6,7\};$$
$$\vartheta_D : \mathcal{I}^D \rightarrow \mathcal{V}^D,\ \vartheta_D(i) = 2^{i+5};$$
$$\mathcal{I}^P = \{1,2,3\};$$
$$\vartheta_P : \mathcal{I}^P \rightarrow \mathcal{V}^P,\ \vartheta_P(i) = (8 + 4 \cdot (i-1))^2.$$

The *index domain constraint*, indicated by the INDEX clause of the ZEN constraint directive, defines a boolean expression over the index domains of the ZEN variables involved.

Definition 3.21. *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 called* index-valid *iff the following condition holds:*

$$valid(\vartheta^{-1}(e_1), \ldots, \vartheta^{-1}(e_n)) \iff \beta\left(\Pi_{j_1,\ldots,j_m}(\vartheta^{-1}(e_1), \ldots, \vartheta^{-1}(e_n))\right) = true,$$

where $\Pi_{j_1,\ldots,j_m}(\vartheta^{-1}(e_1), \ldots, \vartheta^{-1}(e_n))$ *has been defined in Definition 3.18,* $\forall\ \beta : \mathcal{V}^{z_{j_1}} \times \ldots \times \mathcal{V}^{z_{j_m}} \rightarrow$ *boolean an index domain constraint, where:*

$$\{z_{j_1}, \ldots, z_{j_m}\} \subset \{z_1, \ldots, z_n\},\ \forall\ j_k \in [1..n],\ \forall\ k \in [1..m]\ \wedge\ m < n.$$

Informally, a tuple $\left(\vartheta^{-1}(e_1), \ldots, \vartheta^{-1}(e_n)\right) \in \mathcal{I}^{z_1} \times \ldots \times \mathcal{I}^{z_n}$ is index-valid iff it satisfies all the index domain constraints defined across any subset of the ZEN variables involved. An index domain constraint is evaluated by instantiating each ZEN variable z_i with the index of the corresponding ZEN element e_i from the tuple, $\forall\ i \in [1..n]$. All the invalid tuples are eliminated from the multi-dimensional value set.

Example 3.22 (Local index domain constraint).

```
!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, 178] are applications that are executed for different input parameters to examine their effect on the corresponding output results. In a typical scenario, the output parameter values are written to a distinct output file for every experiment. Example 3.22 gives a scenario how ZEN directives can be employed to manage such parameter studies. The local ZEN substitute directives are used to specify the different input and output data

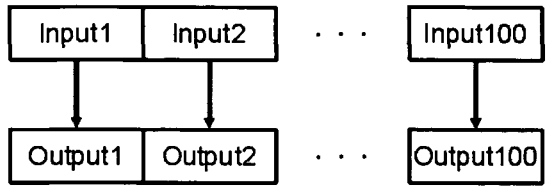


Fig. 3.5. The index domain constraint defined by the Example 3.22.

files to be used in each experiment. The local ZEN constraint directive associates every input file with a correct output file which avoids invalid input and output file combinations (see Figure 3.5):

$$\begin{aligned}
&\mathcal{V}^{\texttt{Input1}} = \{\texttt{Input}i \mid \forall\, i \in [1..100]\},\\
&\mathcal{I}^{\texttt{Input1}} = \{i \mid \forall\, i \in [1..100]\},\\
&\vartheta_{\texttt{Input1}} : \mathcal{I}^{\texttt{Input1}} \rightarrow \mathcal{V}^{\texttt{Input1}},\ \vartheta_{\texttt{Input1}}(i) = \texttt{Input}i;\\
&\mathcal{V}^{\texttt{Output1}} = \{\texttt{Output}i \mid \forall\, i \in [1..100]\},\\
&\mathcal{I}^{\texttt{Output1}} = \{i \mid \forall\, i \in [1..100]\},\\
&\vartheta_{\texttt{Output1}} : \mathcal{I}^{\texttt{Output1}} \rightarrow \mathcal{V}^{\texttt{Output1}},\ \vartheta_{\texttt{Output1}}(i) = \texttt{Output}i;\\
&\mathcal{V}(\texttt{Input1}, \texttt{Output1}) = \{(\texttt{Input}i, \texttt{Output}i) \mid \forall\, i \in [1..100]\}.
\end{aligned}$$

Thus, 100 ZEN file instances are generated (instead of $100 \times 100 = 10000$), each of them reading the data from and writing the data to different input and output files.

3.7.3 Multi-Dimensional Value Set

The following definition redefines the multi-dimensional value set, initially introduced in Definition 3.15, to take the ZEN constraints into consideration.

Definition 3.23. *The* multi-dimensional value set *of a set of ZEN variables $z_1, \ldots, z_n$ is defined as the set of tuples that are both value-valid and index-valid:*

$$\begin{aligned}
\mathcal{V}(z_1, \ldots, z_n) = \{(e_1, \ldots, e_n) \in \mathcal{V}^{z_1} \times \ldots \times \mathcal{V}^{z_n} \mid \\
\mathit{valid}(e_1, \ldots, e_n) \ \wedge\ \mathit{valid}\left(\vartheta^{-1}(e_1), \ldots, \vartheta^{-1}(e_n)\right)\}.
\end{aligned}$$

3.8 ZEN Performance Directive

For performance-oriented program development, the user commonly requires information about the performance of specific code regions, such as the overall execution time, the number of cache misses, the communication time, the synchronisation time, or the floating-point operations per second. The ZEN

language supports the specification of performance metrics to be measured for specific code regions through the *ZEN performance directive.*

In contrast to the other ZEN directives that have general applicability, the ZEN performance directive is only meaningful in the context of parallel applications, following the shared and distributed processing model introduced in Section 2.8.1. The parallel programming paradigms supported are MPI, OpenMP, and HPF.

The scope of the ZEN performance directive can be global for the entire ZEN file or can be limited to a local code region:

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

The ZEN performance directive defines two clauses associated with two sets of mnemonics:

1. *Code region mnemonics (cr_mnem)*, associated with the CR clause, define the code regions within the scope of the directive that are going to be instrumented;
2. *Performance metric mnemonics (pm_mnem)*, associated with the PMETRIC clause, define the performance metrics to be measured for the indicated code regions.

Definition 3.24. *A* code region $\mathcal{CR}$ *is a quadruple that associates a ZEN application* $\mathcal{A}$*, a ZEN file* $\mathcal{Z}$*, a start line number* l_s*, and an end line number* l_e*:*

$$\mathcal{CR} = (\mathcal{A}, \mathcal{Z}, l_s, l_e),$$

where $l_s, l_e \in \mathbb{N}^*$*. A* performance measurement*, denoted by* $\mathcal{M}$*, is an association between a performance metric and a code region:*

$$\mathcal{M} = (pm\_mnem,\ \mathcal{CR}).$$

Let $d \in \mathcal{Z}$ denote a ZEN performance directive that specifies a set of n code regions and p performance metric mnemonics. The set of performance measurements defined by d and denoted as $\mathcal{M}(d)$ is given by to the cross product of the two mnemonic lists:

$$\mathcal{M}(d) = \bigcup_{j=1}^{p} pm\_mnem_j \times \left(\bigcup_{i=1}^{n} \bigcup_{r \in scope(d)} \{r \mid r \text{ is } cr\_mnem_i\} \right).$$

Informally, a global performance directive d collects performance metrics for all the code regions of the ZEN file that contains d. The code region types are specified in the CR clause and the performance metrics in the PERF

clause of d. The local performance directive restricts the performance metrics and the code regions to the corresponding local scope. The local performance measurement directives can also be nested.

The implementation of the ZEN performance directive is based on the SCALEA [161] instrumentation engine and overhead analysis tool built on top of the Vienna Fortran Compiler [15]. SCALEA 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 the OpenMP, MPI, and HPF programming paradigms. A complete list of the code regions and the performance metric mnemonics supported is given in [158].

Example 3.25 (ZEN performance directive).

```
d1:         !ZEN$ CR CR_P, CR_OMPPA PMETRIC WTIME, ODATA
            . . .
CR_OMPPA:   !$OMP PARALLEL NUM_THREADS(4)
            . . .
CR_OMPPA:   !$OMP END PARALLEL
d2:         !ZEN$ CR CR_OMPPA PMETRIC L2_DCM, OCRTP BEGIN
            . . .
CR_OMPDO:   !$OMP PARALLEL DO NUM_THREADS(4)
            . . .
CR_OMPDO:   !$OMP END PARALLEL
d2:         !ZEN$ END CR
```

Example 3.25 is an excerpt of a hybrid OpenMP and MPI parallel program that defines one global ZEN performance directive $d1$, one local ZEN performance directive $d2$, the entire program code region CR_P, and two local OpenMP parallel loops CR_OMPPA and CR_OMPDO. The metrics specified by the two ZEN performance directives are the wallclock time WTIME, the data movement ODATA (i.e., the MPI communication time), the level two data cache misses L2_DCM, and the control of parallelism (i.e., OpenMP fork, join, loop scheduling, and barrier). The following set of performance measurements are generated by this example, given the directive nests displayed:

$$\begin{aligned}\mathcal{M}(d1) = \{&(\texttt{WTIME}, \texttt{CR\_P}), (\texttt{WTIME}, \texttt{CR\_OMPPA}), (\texttt{WTIME}, \texttt{CR\_OMPDO}),\\ &(\texttt{ODATA}, \texttt{CR\_P}), (\texttt{ODATA}, \texttt{CR\_OMPPA}), (\texttt{ODATA}, \texttt{CR\_OMPDO})\};\\ \mathcal{M}(d2) = \{&(\texttt{L2\_DCM}, \texttt{CR\_OMPDO}), (\texttt{OCTRP}, \texttt{CR\_OMPDO})\}.\end{aligned}$$

Definition 3.26. *Let $\mathcal{M}(\mathcal{A})$ denote the set of a performance measurements of a ZEN application defined through ZEN performance directives:*

$$\mathcal{M}(\mathcal{A}) = \{\mathcal{M}(d) \mid \forall\, d \in \mathcal{Z},\ \forall\, \mathcal{Z} \in \mathcal{V}^{\mathcal{A}}\}.$$

An experiment *is a tuple* $(\mathcal{AI}, M)$ *that associates a ZEN application instance* $\mathcal{AI} \in \mathcal{V}^{\mathcal{A}}$ *with a target execution machine* M. *A* performance data *is a function which quantifies each performance measurement for one experiment:*

$$\delta_M : \mathcal{M}(\mathcal{A}) \times \mathcal{V}^{\mathcal{A}} \rightarrow \mathbb{R}.$$

A performance study experiment *is a triplet* $(\mathcal{AI}, M, \delta_M(\mathcal{M}(\mathcal{A}) \times \mathcal{AI}))$ *where* $\mathcal{AI} \in \mathcal{V}^{\mathcal{A}}$ *and* $\delta_M(\mathcal{M}(\mathcal{A}) \times \mathcal{AI}))$ *is the image of the performance data function projected to the subdomain* $\mathcal{M}(\mathcal{A}) \times \mathcal{AI}$.

Informally, a performance study experiment associates an experiment with the complete set of performance data collected from the application, as specified by the complete set of ZEN performance directives.

3.9 Parameter Study Experiment

This section uses the opportunity to define a parameter study experiment as natural side-effect of the formalism presented in this chapter.

Definition 3.27. *An* output parameter *is a tuple* $(\mathcal{Z}_o, pattern)$, *where pattern is a unique pattern that prefixes the output parameter within the output file* $\mathcal{Z}_o$. *Let* $\mathcal{OP}(\mathcal{A})$ *denote the set of output parameters of a ZEN application* $\mathcal{A}$. *An* output data *is a function:*

$$\epsilon : \mathcal{OP}(\mathcal{A}) \rightarrow \mathbb{R}.$$

A parameter study experiment *is a tuple:* $(\mathcal{AI}, Im(\epsilon))$, *where* $\mathcal{AI} \in \mathcal{V}^{\mathcal{A}}$, *and*

$$Im(\epsilon) = \epsilon(\mathcal{OP}(\mathcal{A}))$$

is the image of the output data function.

Definition 3.27 illustrates that the target execution machine of a parameter study experiment is irrelevant.

3.10 The Experiment Generation Algorithm

The ZEN constraints act as a filter over the cross product of all the ZEN variable value sets of a ZEN application. This section introduces an efficient algorithm for generating the valid tuples of ZEN elements, as defined by the multi-dimensional value set in Definition 3.23. The problem is described by the following input and output data of the algorithm:

Input: 1. n ZEN variables: $z_1, \ldots, z_n$ with the value sets $\mathcal{V}^{z_1}, \ldots, \mathcal{V}^{z_n}$;

2. p (value set and index domain) constraints:

$$\gamma_1 : \mathcal{V}_{11} \times \ldots \times \mathcal{V}_{1q},\ \gamma_1 = \textit{log-expr}(\mathcal{V}_{11}, \ldots, \mathcal{V}_{1q}),$$
$$\ldots$$
$$\gamma_p : \mathcal{V}_{p1} \times \ldots \times \mathcal{V}_{pq},\ \gamma_p = \textit{log-expr}(\mathcal{V}_{p1}, \ldots, \mathcal{V}_{pq}),\ \text{where}$$
$$\{\mathcal{V}_{11}, \ldots, \mathcal{V}_{1q}\} \subset \{\mathcal{V}^{z_1}, \ldots, \mathcal{V}^{z_n}\},$$
$$\ldots$$
$$\{\mathcal{V}_{p1}, \ldots, \mathcal{V}_{pq}\} \subset \{\mathcal{V}^{z_1}, \ldots, \mathcal{V}^{z_n}\};$$

Output: $\mathcal{V}(z_1, \ldots, z_n)$.

An algorithm which tests according to Definition 3.23 all the p constraints for all the tuples of the cross product $\mathcal{V}_1 \times \ldots \times \mathcal{V}_n$, has a mathematical complexity of $\mathcal{O}(p \cdot n^o)$, where o denotes the average cardinality of the value sets.

This complexity can be reduced by shifting the focus from the value sets to the ZEN constraints, which are likely to be defined over a smaller subset of ZEN variables. By doing so, the invalid tuples are filtered from the very beginning which avoids further unnecessary and redundant constraint tests. The complexity of the algorithm is reduced to $\mathcal{O}(p\,\overline{n}^o)$, where $\overline{n}$ is represents the average number of ZEN variables in a ZEN constraint logical expression. Obviously, the improvement is due to the fact that $\overline{n} < n$.

Definition 3.28. *Let* $S_{i_1}, \ldots, S_{i_r}, S_{j_1}, \ldots, S_{j_s}$ *denote* $r + s$ *arbitrary sets,* $(v_{i_1}, \ldots, v_{i_r}) \in S_{i_1} \times \ldots \times S_{i_r}$ *and* $(v_{j_1}, \ldots, v_{j_s}) \in S_{j_1} \times \ldots \times S_{j_s}$. *The* composition operation $\otimes$ *between two tuples is defined as follows:*

$$\begin{matrix}(v_{i_1}, \ldots, v_{i_r}) \\ \otimes \\ (v_{j_1}, \ldots, v_{j_s})\end{matrix} = \begin{cases} (v_{k_1}, \ldots, v_{k_t}), & \forall\ S_{i_u} = S_{j_w} \in \{S_{i_1}, \ldots, S_{i_r}\} \cap \{S_{j_1}, \ldots, S_{j_s}\}, \\ & 1 \le u \le r\ \wedge\ 1 \le w \le s,\ v_{i_u} = v_{j_w}, \\ ( ), & \exists\ S_{i_u} = S_{j_w} \in \{S_{i_1}, \ldots, S_{i_r}\} \cap \{S_{j_1}, \ldots, S_{j_s}\}, \\ & 1 \le u \le r\ \wedge\ 1 \le w \le s,\ \textit{such that}\ v_{i_u} \neq v_{j_w}, \end{cases}$$

where:

$$(v_{k_1}, \ldots, v_{k_t}) \in S_{k_1} \times \ldots \times S_{k_t},$$
$$\{S_{k_1}, \ldots, S_{k_t}\} = \{S_{i_1}, \ldots, S_{i_r}\} \cup \{S_{j_1}, \ldots, S_{j_s}\},$$
$$(v_{i_1}, \ldots, v_{i_r}) = \Pi_{i_1, \ldots, i_r}(v_{k_1}, \ldots, v_{k_t}),$$
$$(v_{j_1}, \ldots, v_{j_s}) = \Pi_{j_1, \ldots, j_s}(v_{k_1}, \ldots, v_{k_t}).$$

The composition operator $\otimes$ has the following properties:

1. *commutativity:* $\mathrm{A} \otimes \mathrm{B} = \mathrm{B} \otimes \mathrm{A}$;
2. *associativity:* $\mathrm{A} \otimes (\mathrm{B} \otimes \mathrm{C}) = (\mathrm{A} \otimes \mathrm{B}) \otimes \mathrm{C}$;
3. *idempotency:* $\mathrm{A} \otimes \mathrm{A} = \mathrm{A}$;
4. *neutral element:* $\mathrm{A} \otimes ( ) = \mathrm{A}$;

5. $(\mathcal{G}, \otimes)$ is an *Abelian group*, where $\mathcal{G} = S_{k_1} \times \ldots \times S_{k_t}$.

Lemma 3.29. *Let $z_1, \ldots, z_n$ denote n ZEN variables and let $\gamma_1, \ldots, \gamma_p$ denote p (value set or index domain) constraints over the n ZEN variables (log-expr denotes a logical expression):*

$$\begin{aligned}
&\gamma_1 : \mathcal{V}_{11} \times \ldots \times \mathcal{V}_{1q},\ \gamma_1 = \textit{log-expr}(\mathcal{V}_{11}, \ldots, \mathcal{V}_{1q}),\\
&\qquad\qquad \{\mathcal{V}_{11}, \ldots, \mathcal{V}_{1q}\} \subset \{\mathcal{V}^{z_1}, \ldots, \mathcal{V}^{z_n}\};\\
&\qquad\qquad \ldots\\
&\gamma_p : \mathcal{V}_{p1} \times \ldots \times \mathcal{V}_{pq},\ \gamma_p = \textit{log-expr}(\mathcal{V}_{p1}, \ldots, \mathcal{V}_{pq}),\\
&\qquad\qquad \{\mathcal{V}_{p1}, \ldots, \mathcal{V}_{pq}\} \subset \{\mathcal{V}^{z_1}, \ldots, \mathcal{V}^{z_n}\};
\end{aligned}$$

and

$$\begin{aligned}
&t_1 \in \mathcal{V}_{11} \times \ldots \times \mathcal{V}_{1q},\ \gamma_1(t_1) = \textit{true};\\
&\qquad\qquad \ldots\\
&t_p \in \mathcal{V}_{p1} \times \ldots \times \mathcal{V}_{pq},\ \gamma_p(t_p) = \textit{true}.
\end{aligned}$$

Then $t_1 \oplus \ldots \oplus t_p$ is valid (i.e., valid$(t_1 \oplus \ldots \oplus t_p)$).

Proof. Case 1: $t_1 \oplus \ldots \oplus t_p = (\,)$. The empty tuple is obviously valid.

Case 2: $t_1 \oplus \ldots \oplus t_p \neq (\,)$. Assuming that $t_1 \oplus \ldots \oplus t_p$ is not valid, according to the Definition 3.23 there exists a constraint:

$$\gamma_h : \mathcal{V}_{h_1} \times \ldots \times \mathcal{V}_{h_q} \rightarrow \textit{boolean},$$

such that $\gamma_h(t_h) = \textit{false}$, where $t_h = \Pi_{h_1,\ldots,h_q}(t_1 \oplus \ldots \oplus t_p)$. Since $\gamma_1, \ldots, \gamma_p$ are all the constraints defined, then $\gamma_h \in \{\gamma_1, \ldots, \gamma_p\}$, which contradicts one of the p constraints.

Algorithm: The experiment generation algorithm works according to the workflow depicted in Figure 3.6, where:

- I_1 is the cross product of the value sets of the ZEN variables referred by the constraint γ_1:
$$I_1 = \mathcal{V}_{11} \times \ldots \times \mathcal{V}_{1q};$$
- ...
- I_p is the cross product of the value sets of the ZEN variables referred by the constraint γ_p:
$$I_p = \mathcal{V}_{p1} \times \ldots \times \mathcal{V}_{pq};$$
- E_1 are the valid tuples that fulfil the constraint γ_1:
$$E_1 = \{(e_{11}, \ldots, e_{1q}) \in \mathcal{V}_{11} \times \ldots \times \mathcal{V}_{1q} \mid \gamma_1(e_{11}, \ldots, e_{1q}) = \textit{true}\};$$
- ...

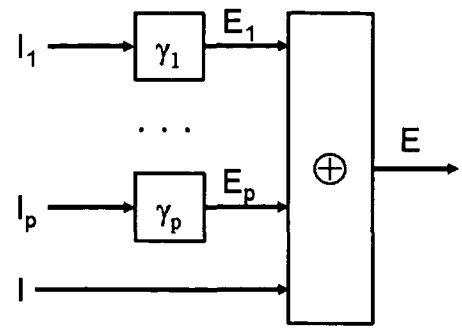


Fig. 3.6. The experiment generation algorithm dataflow.

- E_p are the valid tuples that fulfil the constraint γ_p:

$$E_p = \{(e_{p1}, \ldots, e_{pq}) \in \mathcal{V}_{p1} \times \ldots \times \mathcal{V}_{pq} \mid \gamma_p(e_{p1}, \ldots, e_{pq}) = \mathit{true}\};$$

- I is the multi-dimensional value set of the ZEN variables not referred by any ZEN constraint:

$$I = \{(i_{l_1}, \ldots, i_{l_x}) \in \mathcal{V}^{z_{l_1}} \times \ldots \times \mathcal{V}^{z_{l_x}} \mid \forall\, \gamma_m : \mathcal{V}^{z_{m_1}} \times \ldots \times \mathcal{V}^{z_{m_y}} \rightarrow \mathit{boolean} \wedge \{\mathcal{V}^{z_{l_1}}, \ldots, \mathcal{V}^{z_{l_x}}\} \cap \{\mathcal{V}^{z_{m_1}}, \ldots, \mathcal{V}^{z_{m_y}}\} = \emptyset\};$$

- E is the multi-dimensional value set, obtained by applying the composition operator to the valid tuple elements of the cross product $E_1 \times \ldots \times E_p \times I$:

$$E = \{(e_1, \ldots, e_n) = (e_{11}, \ldots, e_{1q}) \otimes \ldots \otimes (e_{p1}, \ldots, e_{pq}) \mid \forall\, (e_{11}, \ldots, e_{1q}) \in E_1 \wedge \ldots \wedge (e_{p1}, \ldots, e_{pq}) \in E_p\}.$$

Based on the Lemma 3.29, the tuples belonging to the set E are valid, therefore, $\mathcal{V}(z_1, \ldots, z_n) = E$.

Example 3.30 (Constraint evaluations).

```
!ZEN$ ASSIGN A = { 1 : 100 }
!ZEN$ ASSIGN B = { 1 : 100 }
!ZEN$ ASSIGN C = { 1 : 100 }
!ZEN$ CONSTRAINT VALUE A == B
!ZEN$ CONSTRAINT VALUE B == C
```

Example 3.30 defines three ZEN variables with the same value set:

$$\mathcal{V}^A = \mathcal{V}^B = \mathcal{V}^C = \{i \mid \forall\, i \in [1..100]\}.$$

The dataflow multi-dimensional sets computed by the experiment generation algorithm according to the Figure 3.6 are as follows:

$$I_1 = \mathcal{V}^A \times \mathcal{V}^B;$$
$$I_2 = \mathcal{V}^B \times \mathcal{V}^C;$$
$$I = \phi;$$
$$E_1 = \{(a,b) \mid \forall\, a \in \mathcal{V}^A \ \wedge\ \forall\, b \in \mathcal{V}^B \ \wedge\ a = b\};$$
$$E_2 = \{(b,c) \mid \forall\, b \in \mathcal{V}^B \ \wedge\ \forall\, c \in \mathcal{V}^C \ \wedge\ b = c\};$$
$$E = \{(a,b,c) \mid \forall\, a \in \mathcal{V}^A \ \wedge\ \forall\, b \in \mathcal{V}^B \ \wedge\ \forall\, c \in \mathcal{V}^C \ \wedge\ a = b = c\},$$

where:

$$(a,b) \otimes (b,c) = (a,b,c),\ \forall\, a \in \mathcal{V}^A \ \wedge\ \forall\, b \in \mathcal{V}^B \ \wedge\ \forall\, c \in \mathcal{V}^C.$$

Since $|I_1| = |I_2| = 10^4$ and $|E_1| = |E_2| = 10^2$, the experiment generation algorithm evaluates $3 \cdot 10^4$ constraints. In contrast, a straight-forward algorithm, which evaluates both ZEN constraints on all the tuples of the cross product $A \times B \times C$ according to the Definition 3.23, performs $2 \cdot 10^6$ constraint evaluations.

3.11 On-line Application Analysis

The performance and parameter study experiments described in Sections 3.8 and 3.9 assume that the performance and the output parameter data is available *post-mortem* (or *off-line*) after the experiments have completed. This restriction is often critical for the users who need to get access to intermediate data values on-the-fly, as the experiments progress. The Grid computing that defines applications running on unreliable resources is especially prone to such situations, for instance in a typical application steering scenario.

Often users are interested in being notified of important *events* that are specific to their application, e.g., when a certain variable changed its value or when a specific performance metric exceeded a critical threshold. To meet this requirement, the ZEN language has been extended with *event directives* for the specification and the collection of on-line events and data from the running experiments. The event directives proposed in this section are part of a more general event framework which will be presented in detail in Section 5.4.

3.11.1 ZEN Event Directive

Using the *ZEN event directive*, the user can request to be informed of well-defined application run-time status. The ZEN event directive has the following syntax:

zen-event **is** EVENT *ident* [FILTER *bool-expr*] [SAMPLE *rate*]
ident **is** *string*

The directive defines the following three clauses:

1. EVENT defines the event identifier *ident* which must be an arbitrary unique string for an application;
2. FILTER is an optional clause that filters the events to those which satisfy the associated boolean expression. The syntax of the filtering condition defined over a set of program variables is identical to the one defined by the ZEN constraint directive in Section 3.7. The directive assumes no semantic analysis to examine whether the program variables referred by the boolean expression are valid within the run-time evaluation scope of the filtering condition. An eventual *"variable not found"* error will be produced by a subsequent ZEN file compilation.
3. SAMPLE is an optional clause which determines the directive applicability mode, as follows:
   a) *procedural mode* is selected by omitting the SAMPLE clause. Whenever the program counter reaches the directive at runtime, an event of type *ident* is generated if the filtering condition yields true. The variables involved in the filtering condition must be valid within the scope where the directive is defined;
   b) *threaded mode* is selected by introducing the SAMPLE clause which specifies the rate (in samples per second) at which the filtering condition shall be evaluated. If the filtering condition yields true, an event of type *ident* is generated. In the threaded mode the variables involved in the filtering condition must be global.

Example 3.31 (ZEN event directive).

```
!ZEN$ EVENT N1000 FILTER N > 1000 SAMPLE 1
```

Example 3.31 defines a ZEN event directive operating in the threaded mode, which generates an event of the type N1000 if the program variable N is greater than 1000. The variable N must be global and is sampled every second.

3.11.2 ZEN Performance Directive

For on-line performance analysis of parallel applications, the ZEN performance directive introduced in Section 3.8 has been extended with three extra clauses for expressing performance events:

global-perf **is** CR *cr_mnem-list* PMETRIC *pm_mnem-list*
[EVENT *ident*] [SAMPLE *rate*] [FILTER *bool-expr*]
local-perf **is** CR *cr_mnem-list* PMETRIC *pm_mnem-list*
[EVENT *ident*] [SAMPLE *rate*] [FILTER *bool-expr*] BEGIN
code-region
END CR

The semantics of the three additional (and also optional) clauses are as follows:

1. EVENT defines the event type;
2. SAMPLE is an event parameter which defines the rate at which the performance metrics specified by the directive are periodically sampled. No sampling is done if this clause misses (i.e., post-mortem analysis). The measurement unit is samples per second. For each measurement, an event of type *ident* is generated if the boolean expression specified by the FILTER clause yields true (or misses). The sampling rate defines the expiration time of each event;
3. FILTER defines a filter as a boolean expression over the performance metric mnemonics specified by the directive. The performance mnemonics referred by the boolean expression must be present within the PMETRIC clause. An expression evaluation occurs at run-time at the rate specified by the SAMPLE clause. If the expression yields true, an event of type *ident* is generated.

The implementation of the online clauses of the ZEN performance directive uses the Process Manager sensor for the dynamic instrumentation of running processes, which will be described in Section 5.2.2.

Example 3.32 (Online ZEN performance directive).

```
!ZEN$ CR CR_P PMETRIC ODATA, WTIME EVENT comm SAMPLE 4
              FILTER ODATA > WTIME / 2
```

Example 3.32 illustrates a global ZEN performance directive which measures the execution time and the communication time of the entire program. The two metrics are sampled four times per second. An event of type comm is generated if the communication overhead ODATA dominates (i.e., is greater than half of) the wallclock execution time WTIME.

4

ZENTURIO Experiment Management Tool

ZENTURIO [124, 128] is a tool designed to automatically generate and conduct large number of experiments in the context of large scale performance and parameter studies on cluster and Grid architectures. ZENTURIO has been designed as a distributed service architecture illustrated in Figure 4.1, compliant with the service-oriented Grid infrastructure model presented in Chapter 2.

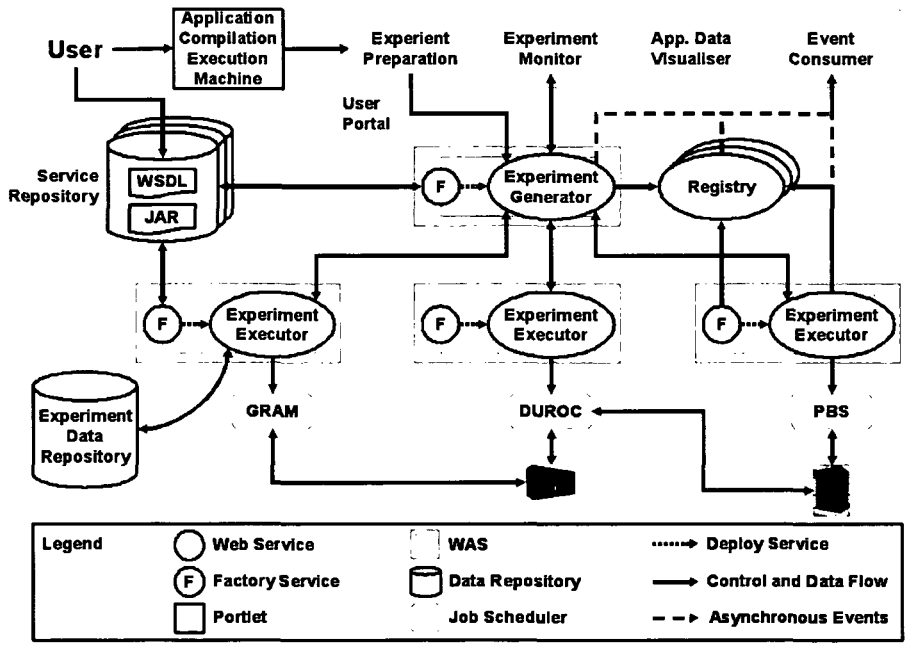


Fig. 4.1. The ZENTURIO experiment management tool architecture.

ZENTURIO uses the ZEN directive-based language defined in Chapter 3 to annotate arbitrary application files and specify value ranges for any problem, system, or machine parameter, including program variables, file names, compiler options, target machines, machine sizes, scheduling strategies, and data distributions. The functionality of the ZENTURIO experiment management tool described in this chapter is restricted to post-mortem cross-experiment performance analysis and parameter studies. The ZEN performance directives introduced in Section 3.8 are used to indicate the performance metrics to be measured and computed.

The entry point for a user is a graphical *User Portal* (see Section 4.1) which normally resides on the local client machine (e.g., laptop). Through the portal, the user creates or loads a ZEN application which is subject to a large-scale performance and parameter study automatically conducted by ZENTURIO.

The experiment management functionality of ZENTURIO is achieved through the cooperative use of various distributed Grid services. The *Service Repository* (see Section 5.3.2) is a database that contains persistent implementations of Grid services. The *Factory* (see Section 5.3.4) is a service in charge of creating service instances on arbitrary Grid sites using the implementation information from the Service Repository. The *Registry* (see Section 5.3.5) manages an up-to-date list of existing transient Grid service instances and provides a variety of advanced high-throughput service discovery operations. The Service Repository, the Factory, and the Registry are generic Grid services that are fundamental to the tool integration framework which will be addressed in Chapter 5.

After a ZEN application has been properly input, ZENTURIO automatically generates, executes, controls, and monitors the experiments on the target Grid site automatically. The User Portal uses the Registry to locate an *Experiment Generator* service, preferably on the local Grid site. If the Experiment Generator resides on a different site, the application files are packed into a ZIP [45] archive and sent to the destination site using the GridFTP protocol. If no Experiment Generator service is found, an instance is created using the Factory service. The Experiment Generator parses the ZEN files, instruments the application according to the ZEN directives encountered, and generates the corresponding set of experiments.

After having generated one experiment, the Experiment Generator transfers it to the target execution Grid site, where an *Experiment Executor* service resides. If no Experiment Executor service is found, an instance is created using the Factory service. The Experiment Executor is a generic service responsible for compiling, executing, and managing the execution of multiple experiments. Upon the completion of each experiment, the Experiment Executor automatically stores the experiment output and the performance data into a well-defined PostgreSQL [83]-based *Experiment Data Repository* (see Section 4.4). The users can remotely access the data stored in the reposi-

tory via the User Portal or manually formulate SQL queries for post-mortem performance analysis and visualisation.

It is usual in practice that the end-users cannot stay online for the entire duration of the performance or parameter study, for instance when submitting a large suite of experiments over night or when travelling. For this reason, the ZENTURIO architecture has been designed such that the Experiment Generator is the only service with which the User Portal interacts. Once the user has submitted a ZEN application, the Experiment Generator maintains the complete information about the application and the associated experiments. This allows the users to *disconnect* the portals from the Grid without loosing the contact information to their experiments. The users can subsequently open the portal at any time from arbitrary Grid locations, *connect* to the Experiment Generator, retrieve the status of the experiments, and perform the desired performance analysis or the parameter visualisation studies.

All the methods of the Experiment Executor and the Experiment Generator services are provided in both synchronous (blocking) and asynchronous (non-blocking) mode. Asynchronous methods return an asynchronous receipt, on behalf of which synchronous methods can be invoked to poll for available results. Such asynchronous methods, which are part of the general event framework presented in Section 5.4, are crucial for implementing highly-responsive clients that do not block upon calling long running synchronous methods. All the services provided by ZENTURIO, as well as the Experiment Data Repository can be *accessed concurrently* by multiple clients, which is a key feature for providing scalable Grid infrastructures.

4.1 User Portal

The *User Portal* is a client application that enables user friendly graphical access to the functionality provided by ZENTURIO. The User Portal is a small light-weight program easy to install and manage on the local machine (e.g., laptop), which hinders the end-users by the complexity from the underlying Grid environment.

The User Portal can operate in three modes:

1. *Online Grid* is the standard mode of operating in a Grid infrastructure, as presented at the beginning of this chapter. The user must first authenticate using the GSI credentials (see Secure 2.4), which returns a limited proxy required for secure communication with any remote Grid service. GRAM [38] and DUROC [39] are the job schedulers used in this mode;
2. *Online Cluster* accommodates a simplified instance of the ZENTURIO infrastructure on the local cluster front-end. The Grid services are replaced by ordinary Java objects, while the GSI security comprising the user authentication are disabled. The local job scheduler (e.g., [108, 91, 179, 166, 152] that manages the cluster nodes is used in this mode;

3. *Offline* employs the User Portal for post-mortem analysis and visualisation based on the data already stored into the Experiment Data Repository (i.e., by previous experiments executed by ZENTURIO in the online mode).

The User Portal consists of four portlets for interacting with the user which will be described in the following sections. A snapshot of the User Portal main frame conducting a real application is depicted in Figure 4.2.

4.1.1 ZEN Editor

The *ZEN editor* provides a user friendly graphical interface that facilitates the annotation of ZEN files with ZEN directives by hiding syntactic language details (e.g., escape \ characters). The local directive scopes (i.e., of substitute, constraint, and performance directives) are easily indicated through mouse-based code region selection. An important task of the ZEN editor is to provide a centralised display of all the directives inserted in various ZEN files. Additionally, the total number of experiments implied by the ZEN directives inserted is provided. This online information is useful for tuning the application parameter space to a reasonable size before generating the full set of experiments. A snapshot of the ZEN editor is illustrated in Figure 4.3.

4.1.2 Experiment Preparation

The *Experiment Preparation* portlet of the User Portal depicted in Figure 4.4 assists the user in the specification of a suite of experiments for performance or parameter study purposes through the following inputs:

1. *ZEN application* represented by a list of files which can are individually selected from arbitrary local directories. These are categorised into:
   a) *ZEN files* to be processed by the ZEN Transformation System (see Section 3.3), which are further classified as follows:
      i. *ZEN source files* that require performance instrumentation using the SCALEA instrumentation engine based on the source-to-source Vienna Fortran Compiler [15]. The implementation is therefore limited to Fortran 90 source files. Additionally, a ZEN source file instance requires the compilation of the ZEN application instance using a back-end (Fortran 90) compiler.
      ii. *ZEN script files* or input files that do not require any compilation of the ZEN application instance. This information is used by the Experiment Executor in optimising the compilation of the entire experiment suite.
   b) *Regular files* that do not contain ZEN directives and therefore are not processed by the ZEN Transformation System;

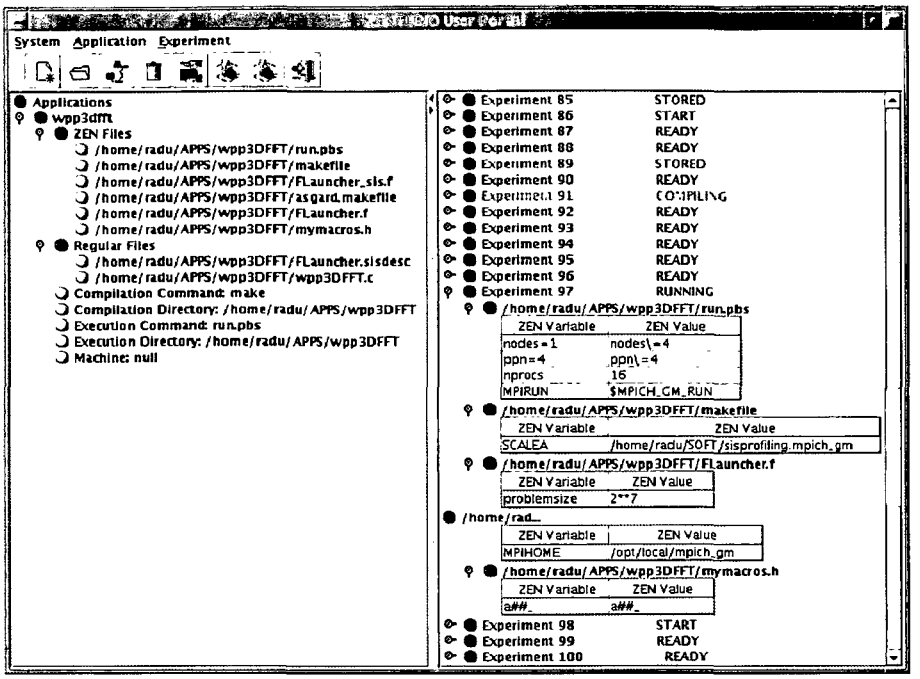


Fig. 4.2. A snapshot of the User Portal.

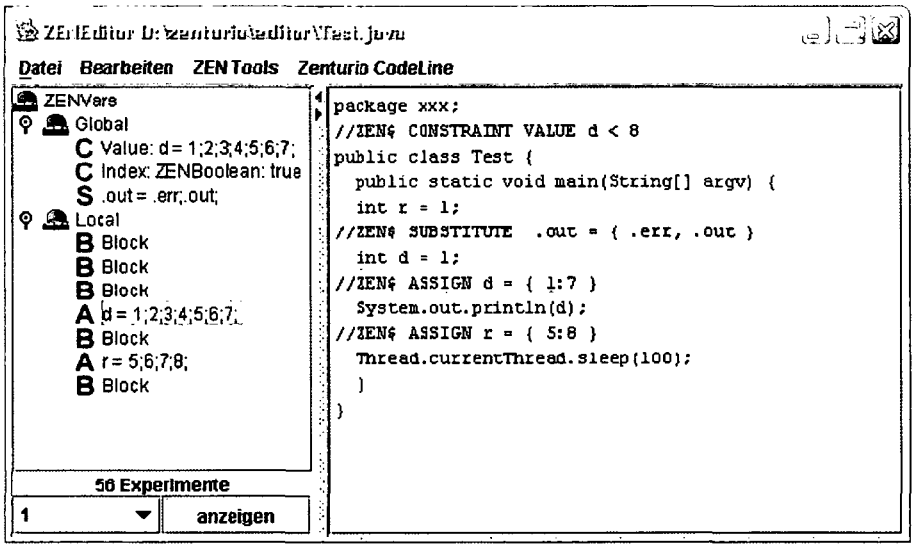


Fig. 4.3. A snapshot of the ZEN editor.

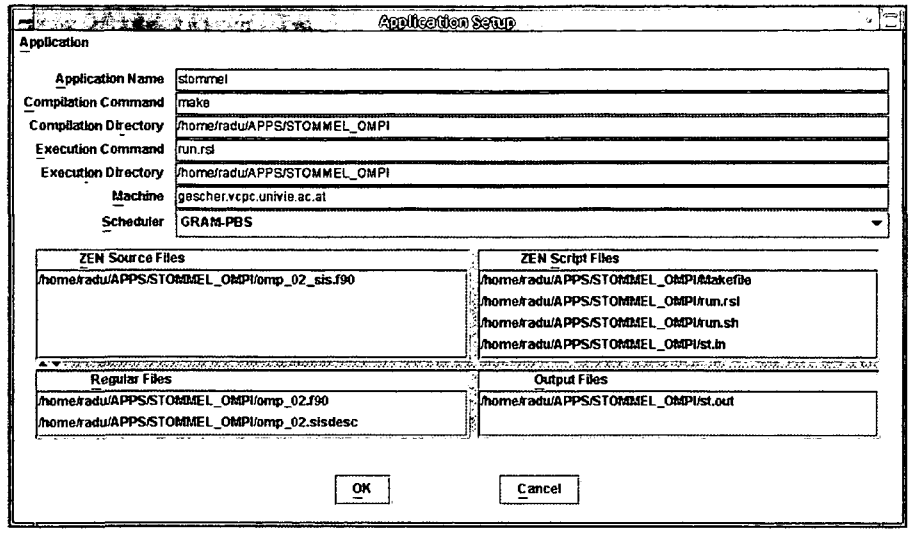


Fig. 4.4. A snapshot of the Experiment Preparation portlet.

Input and output file staging in the online Grid mode is achieved through the GASS [18] functionality and is automatically handled by the GRAM [38] resource manager;
2. *compilation directory* and the *compilation command*;
3. *execution directory* and the *execution command*;
4. *Grid site* where to execute the experiments;
5. back-end batch *scheduler* to be used;
6. *output files* for parameter study purposes.

After receiving all the inputs, the Experiment Preparation portlet automatically contacts the Registry for an available Experiment Generator service (preferably on the local site or on the execution site to minimise file transfers) that generates the experiments. If no Experiment Generator service is found, a transient instance is created using the Factory. If the Experiment Generator resides on a remote Grid site, the ZEN application is packed in a ZIP archive and transferred using the GridFTP protocol.

4.1.3 Experiment Monitor

The *Experiment Monitor* portlet uses the Experiment Executor service to remotely compile, execute, control, and monitor experiments running on the target Grid site. The user interface of the Experiment Monitor is displayed in the right panel of the User Portal depicted in Figure 4.2. Upon the selection of a ZEN application in the Experiment Preparation left panel, the corresponding set of experiments are automatically displayed in the right panel. The experiments of a ZEN application can be submitted for execution either

individually, or on a collective basis. Each experiment is displayed accompanied by its status highlighted using a different colour. Upon clicking on an experiment, all the ZEN variable instantiations that describe the experiment are expanded. A *filtering* capability allows the user to select, display, or search for a subset of experiments according to specific ZEN variable instantiations.

As part of the Experiment Monitor, an *event listener* (thread) receives notifications from the Experiment Executor about changes in the status of individual experiments. This is a light-weight and highly responsive mechanism for providing a consistent up-to-date view of the generated experiments and their status, which avoids unnecessary expensive polling. This functionality is part of the more general ZENTURIO event framework, which will be presented in detail in Section 5.4.

Figure 4.5 displays the state transition diagram of an experiment executed with ZENTURIO. The state diagram has one initial state *start* and two final states *stored* and *fail*. After being created by the Experiment Generator, the experiment is initialised in the *start* state. If the Experiment Executor site is different from the Experiment Generator site, the experiment goes through the optional *transfer* state, during which it is copied to the target execution site. If an experiment (i.e., the associated application instance) needs compilation after being copied to the execution site, it goes through the *compiling* state. If the experiment is part of a binary (already compiled) ZEN application, it skips the *compiling* state and goes directly into the *ready* state. The *ready* state specifies that the experiment is ready for execution. From this state, the experiment can go either into the *waiting* state, if the execution is postponed (e.g., through reservations), or into the *queued* state, if the experiment is submitted to a batch job scheduler. If the experiment is forked, it goes directly into the *running* state. After the experiment has completed, the state changes to *terminate*. The final state *stored* indicates that the experiment (including the output files and the performance data) has been stored into the Experiment Data Repository. If an erroneous operation takes place (e.g., compilation or execution error) during any of the states or if the experiment is explicitly killed, the experiment goes in to the *failed* state. From the *terminated*, *stored*, and *failed* states an experiment can change to the *ready* state, if reexecution is desired (e.g., in case of casual non-deterministic faulty executions).

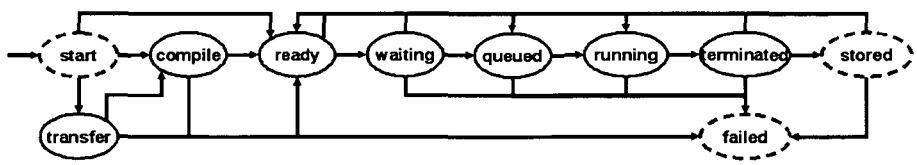


Fig. 4.5. The experiment state transition diagram.

4.1.4 Application Data Visualiser

ZENTURIO defines two types of output data for an experiment which are automatically stored into the Experiment Data Repository for post-mortem analysis:

1. *performance metrics* (e.g., execution time, synchronisation, communication) which are specified through ZEN performance directives (see Section 3.8);
2. *output results* which are retrieved from the output files indicated by the user in the experiment preparation phase (see Section 3.9).

The *Application Data Visualiser* portlet has been designed for post-mortem analysis through automatic query of the performance and the output data stored in the Experiment Data Repository. The data can be either tabulated into ASCII files or graphically represented using on the ASKALON visualisation package [55] which provides various linechart, barchart, piechart, and surface diagrams.

Figure 4.6 shows a snapshot of the Application Data Visualiser for constructing a cross-experiment performance visualisation diagram. The top-left panel displays the list performance metrics computed (e.g., barrier, collective communication, control of parallelism) which can be selected for visualisation. The performance metrics can be organised in two different tree-based visualisation hierarchies:

1. *Metric-to-Region* (shown in Figure 4.6) displays on the first tree level the complete list of the metrics computed. The next tree levels below the metric level display the region hierarchies for which each parent metric holds;
2. *Region-to-Metric* displays the complete hierarchy of code regions for which the performance metrics have been collected. The leaves of the tree represent the performance metrics which have been measured for the parent (sub-)region.

Upon the mouse selection of a metric, the top-right panel of the Application Data Visualiser dialog-box displays the affiliated source code region (if this information is available). The bottom panel displays the complete set of ZEN variables that annotate the ZEN application. Every ZEN variable has an associated list box that contains its complete value set. To generate a visualisation, the user must select a subset of experiments and map ZEN variables to visualisation axis by instantiating ZEN variables with appropriate ZEN elements. The mapping of the ZEN variables to the visualisation axis is obtained by introducing two special ZEN elements to each value set:

1. *Wildcard* indicates that the ZEN variable is selected as a visualisation axis. The ZEN elements of the ZEN variable are displayed on the axis in the order given by the index domain function (see Definition 3.20). A number of n wildcard selections define an $n+1$-dimensional visualisation.

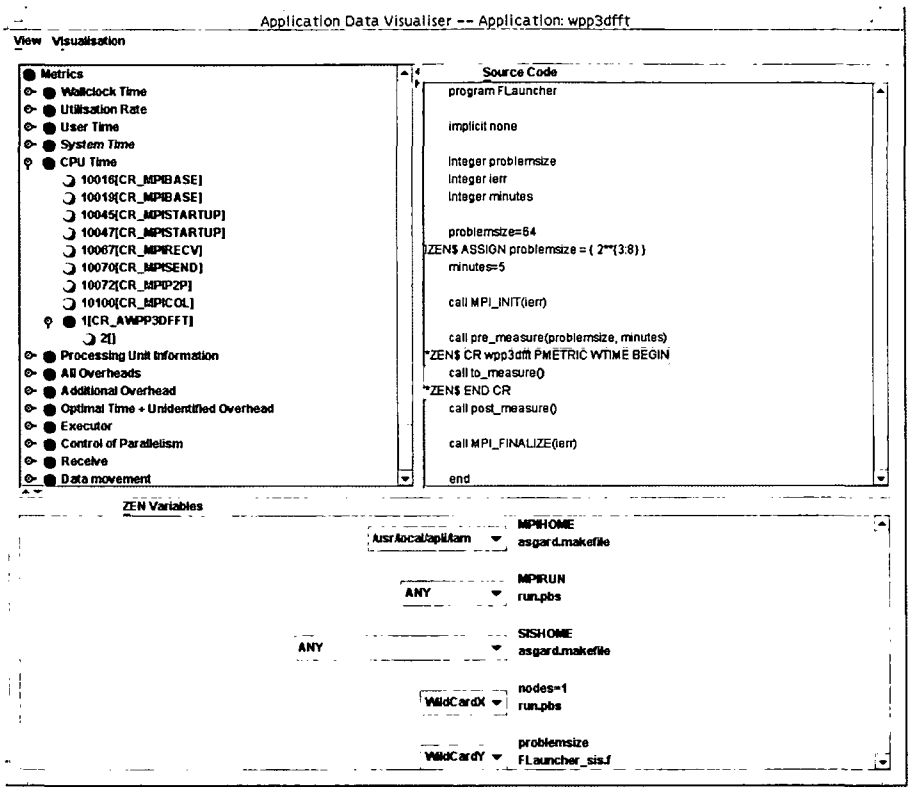


Fig. 4.6. A snapshot of the Application Data Visualiser for performance studies.

The current implementation is limited to three wildcard ZEN variables, but can be easily extended. To exactly control the axis onto which the values of a wildcard ZEN variable are mapped, three wildcard flavours are supported: *WildcardX*, *WildcardY*, and *WildcardZ* (i.e., the ZEN variable is displayed on the X, Y, respectively Z axis);

2. *ANY* matches any value and indicates that the ZEN variable is irrelevant for the visualisation and should be ignored. A typical case for an ANY selection is when the ZEN variable is bound via a ZEN constraint to another ZEN variable which has received a wildcard.

The X, Y, and Z axis names are pre-defined for each ASKALON visualisation diagram. Single or multiple selection of metric-region pairs is allowed for visualisation. Upon a single metric selection, the metric represents one fixed pre-defined axis in the visualisation diagram. Upon a multiple metric selection, the metric visualisation axis must be indicated through a metric wildcard selection in the dialog-box menu (i.e., *Visualisation* menu item). If no metric wildcard is indicated, only the last selected metric is visualised.

Figure 4.7 displays a similar Application Data Visualiser dialog-box used for parameter study purposes to visualise the output results across multiple experiments. The performance metric panel is replaced with a list of application output files, which includes the standard output and the standard error streams. An output result is specified by selecting an output file and introducing a unique pattern that prefixes the output result within the output file, as formally specified in Section 3.9. This pattern is used to extract the output result from the output file of each experiment involved in the visualisation.

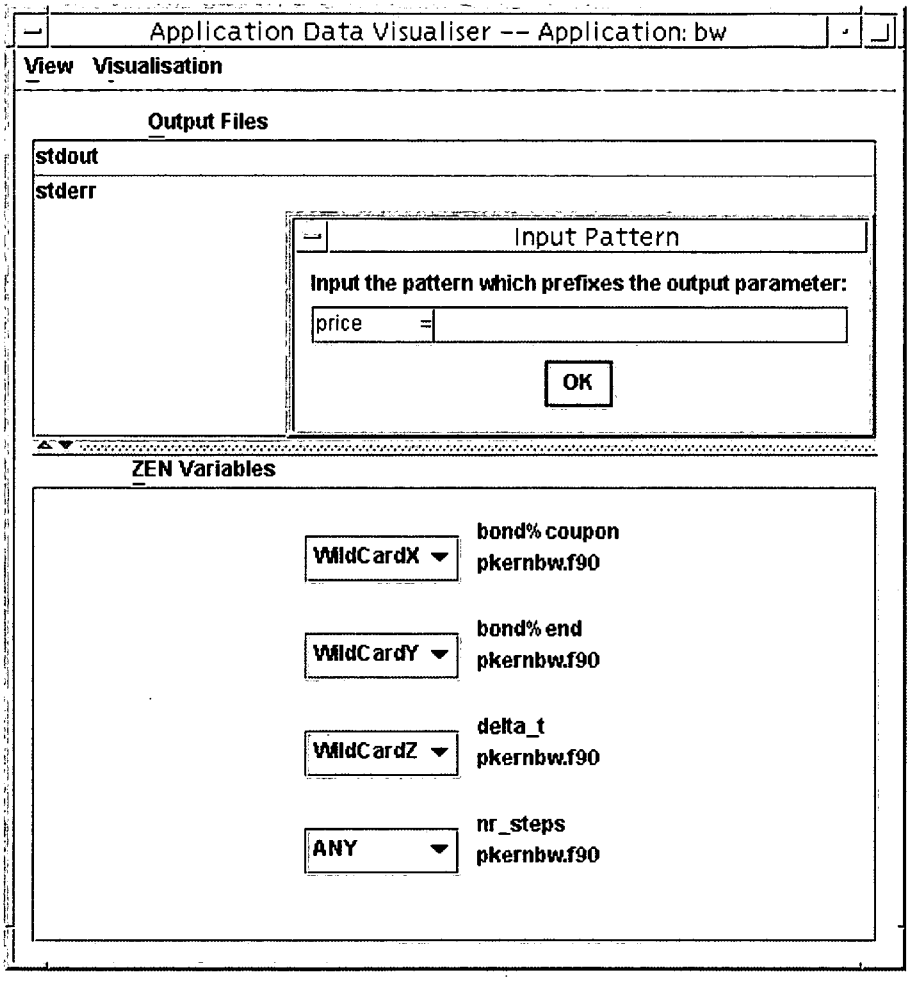


Fig. 4.7. A snapshot of the Application Data Visualiser for parameter studies.

4.2 Experiment Generator

The Experiment Generator is a Grid service in charge of generating the experiments from an input ZEN application, as specified in Chapter 3. The architecture of the Experiment Generator is displayed in Figure 4.8. Each ZEN file of the ZEN application is first parsed using the scanner and parser modules of the ZEN Transformation System which produce an *abstract syntax tree* as presented in Section 3.3. The abstract syntax trees of all the ZEN files are given as input to the ZEN Constraint Evaluation Algorithm which generates the set of valid ZEN element tuples, as presented in Section 3.10. The valid ZEN element tuples determine the set of valid ZEN application instances, whose constituent ZEN file instances are generated using the unparser module of ZEN Transformation System. A ZEN application instance is the foundation of an experiment, as formally defined in the Definitions 3.26 and 3.27.

The SCALEA [159] instrumentation engine, which provides a complete Fortran 90 OpenMP, MPI, and HPF front-end and unparser, is used to instrument the application for performance metrics based on the ZEN performance directives. The Experiment Generator has been designed as a separate service to isolate the platform dependencies and the proprietary components of the Vienna Fortran Compiler [15] on which the SCALEA instrumentation engine is based. The Experiment Generator typically runs as a pre-installed Grid service that serves remote experiment generation requests through a portable platform-independent API.

The Experiment Generator provides an API to logically insert ZEN directives into the abstract syntax tree of each parsed ZEN file in cases when it is not practical to insert the directives manually. This feature is important when a large number of ZEN variables are required to annotate the ZEN application, for instance in the case of large-scale Grid scheduling problems which will be discussed in Sections 6.2 and 6.4.

The Experiment Generator provides four methods for generating experiments for a ZEN application:

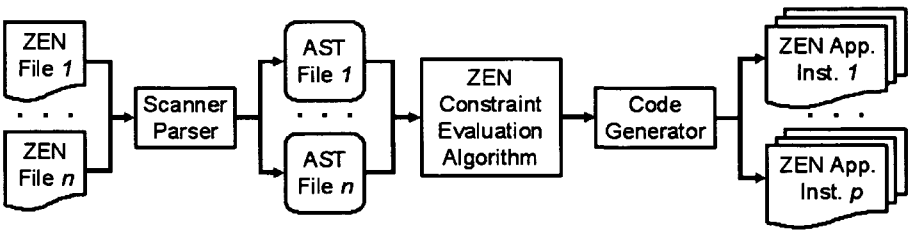


Fig. 4.8. The Experiment Generator architecture.

1. *synchronous*, by means of a single method call. This approach is rather primitive since the synchronous invocation can be very expensive and produces blocking (i.e., non-responsive) clients;
2. *iterative*, the next experiment being returned by an iterator upon synchronous request. This method is compliant with the pull event model (see Section 5.4);
3. *asynchronous*, each experiment being sent to the client using an asynchronous callback as soon as it is generated. This method is compliant with the push event model (see Section 5.4);
4. *random*, by instantiating each ZEN variable (or a subset of them) with a random ZEN element. This method is required by the optimisation heuristics that will be presented in Chapter 6.

Additionally, for Grid applications submitted using the GRAM or DUROC job schedulers (i.e., ZENTURIO running in online Grid mode), the Experiment Generator transfers the experiments to the target Grid execution sites using the GridFTP protocol. In the case of DUROC, the experiments are copied to multiple destination Grid sites, which are read from the RSL description of the application.

4.3 Experiment Executor

The Experiment Executor is a generic service with a high-level interface for executing and managing experiments on target Grid execution sites. The Experiment Executor has been designed as a stand-alone Grid service independent of ZENTURIO that can be deployed for experiment management purposes within other infrastructures too. The Experiment Executor assumes a properly installed application on the target execution site(s).

The current implementation supports interfaces to the following batch job schedulers:

1. *fork* [148] for single processor machines which host both the Experiment Executor service and the running experiments.
2. *Condor [108], LoadLeveler [91], Load Sharing Facility (LSF) [179], Maui [34], Portable Batch System (PBS) [166], Sun Grid Engine (SGE) [152]* for dedicated workstation clusters. This configuration is employed by ZENTURIO in the online cluster mode. The Experiment Executor resides on the cluster front-end and must receive a job submission script compliant to the job batch scheduler used to execute the cluster experiments;
3. *GRAM* [38] and *DUROC* [39] for executing remote experiments on a single, respectively multiple Grid sites. This configuration is employed by ZENTURIO in the online Grid mode. The Experiment Executor may reside on arbitrary Grid sites and must receive an RSL script to execute the experiments.

The Experiment Executor provides functionality to:

- add and remove experiments;
- compile experiments;
- execute experiments;
- retrieve the status of experiments;
- subscribe for experiment status change notification callbacks, according to the push event model that avoids the polling overhead (see Section 5.4).
- terminate experiments;
- stage-in input data files from specific Grid sites;
- stage-out experiment output to indicated Grid sites (i.e., standard output, standard error, any output file, and performance data);
- retrieve all the experiments associated with a certain application (optionally restricted to a certain state);
- set the maximum number of experiments that are concurrently executed. This feature allows the user to restrict the number of experiments simultaneously submitted to the cluster queue to a decent pre-defined number, or to control the number of experiments concurrently forked on the same (SMP) machine (i.e, normally one on single processor machines);
- the number retries in case of faulty executions. This feature is a crucial for improving the fault tolerance, as often the execution of large number of experiments on cluster and Grid architectures is prone to non-deterministic failures due to unpredictable underlying resource management support;
- store the experiment-specific data (i.e., ZEN variables and ZEN elements), output files, and performance data into the Experiment Data Repository.

All the operations provided by the Experiment Executor service can be applied on individual or collective basis, by providing appropriate input filters (e.g., all the experiments belonging to an application).

4.4 Experiment Data Repository

ZENTURIO stores post-mortem information about the ZEN application and the associated experiments into a common *Experiment Data Repository.* The Experiment Data Repository has been designed as a relational database implemented on top of PostgreSQL [83]. Upon the completion of each experiment, the Experiment Executor stores the descriptive information about the experiment (i.e., the ZEN variable instantiations), the standard output, the standard error, the performance data, and the output files, depending on the experiment type. In the case of large output files, the URL location to the GASS file system is stored. The Experiment Data Repository enables various users and tools to interoperate by exchanging post-mortem performance and output data from previous experiments. Figure 4.9 displays the UML diagram that models the Experiment Data Repository relational schema.

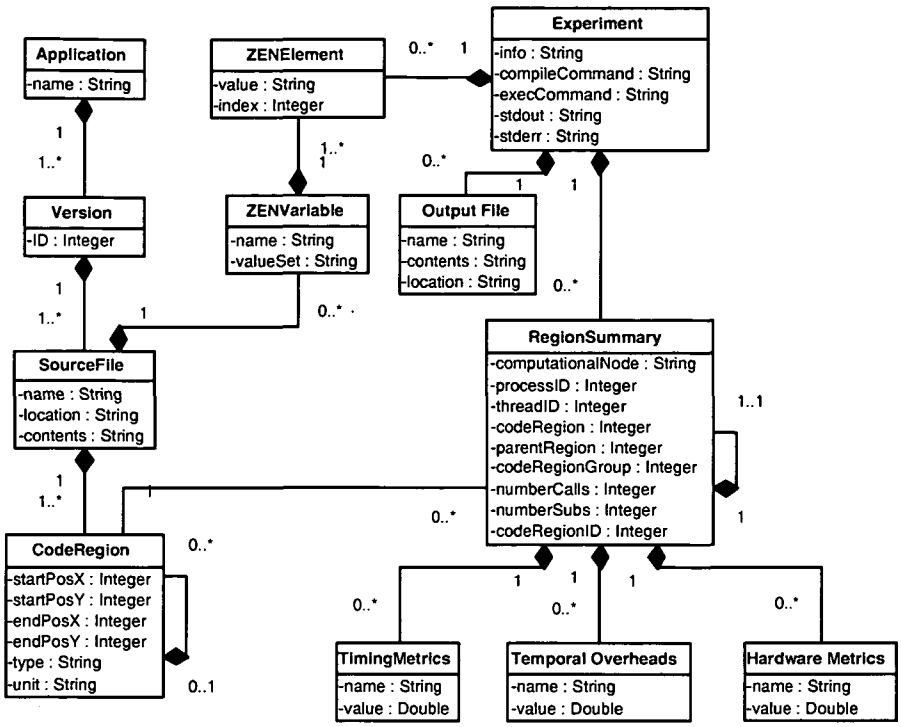


Fig. 4.9. The Experiment Data Repository schema definition.

5

Tool Integration

As applications get larger and more complex, the use of software tools becomes vital for tuning application parameters, identifying performance leaks, or detecting program defects. Extensive efforts within academia and industry over the last decade have resulted in a large collection of tools for practical application engineering. Available tools of broad interest include program source and structure browsers, editors, static program analysers, performance predictors, optimisation compilers, execution control and monitoring environments, sequential and parallel debuggers (providing deadlock detection and deterministic message replay mechanisms), data and execution visualisers, performance analysers, or various program tracers.

Despite of all these huge efforts in the tool development to ease the parallel program development, the user acceptance in the scientific community has not been achieved. Most users still base their application development activities on manual source program instrumentation and a tedious, error-prone, and time consuming *instrumentation – compilation – link – execution – data collection – data analysis* cycle. There are two reasons for this unfortunate situation:

1. *Portability.* Most of the existing application tools are not available on multiple parallel platforms, primarily because of their limited portability. When using a new parallel system the user must in most cases learn and familiarise with new tools with different functionality and user interfaces. This requires additional (often unnecessary) time and effort and can be a major deterrent against the use of more appropriate computer systems.
2. *Interoperability.* Most of the tools cannot be used cooperatively to further improve programming efficiency, mainly because they are insufficiently integrated into a single coherent environment. Existing integrated tool environments [33, 175] comprising several tools do offer some degree of interoperability. They do, however, have the disadvantage that the set of tools provided is fixed, typically decided by the initial project objectives. The resulted tools interact through internal proprietary interfaces which can not easily be extended. The outcome is in fact not an interop-

erable tool-set, but a more complex monolithic tool which combines the functionality of the integrated tools, but lacks true interoperability and extensibility.

Based on the type of analysis performed, one can distinguish between two types of software tools:

1. *Offline tools* completely separate the run-time data collection from the data analysis phase. Run-time data analysis is typically performed post-mortem after the application has completed. The ZENTURIO experiment management tool presented in Chapter 4 is a typical offline tool example.
2. *Online tools* collect and analyse the data on-the-fly during the execution of the application using special purpose *monitoring systems.*

There are two fundamental reasons why most of the run-time tools cannot be cooperatively used by the program developer on the same application:

1. Run-time tools use different instrumentation techniques. While offline tools can easily solve this problem by means of standardised trace data formats [76] or common data repositories [60, 160], online tools suffer from incompatible complex run-time monitoring systems. Most tools require special preparation of the application with specialised compilation and link flags, which leads to undesired conflicts and makes the interoperability impossible.
2. At inception, tools are not considered or designed for interoperability. Most tools are designed and built as stand-alone applications and can only be used in isolation. Tool interoperability is a complex issue that has to be considered as a major objective when the tools are first designed and cannot simply be added as an afterthought.

The thesis solves the offline tool interoperability problem by proposing a common Experiment Data Repository for sharing performance data, as described in Section 4.4. The reminder of this chapter focuses on the online tool interoperability problem.

5.1 Design

The ZENTURIO architecture presented in Chapter 4 has been designed in the context of a more general tool integration framework depicted in Figure 5.1. The framework [97, 98, 131] defines a three-tier service-oriented architecture for interoperable tool development that instantiates the abstract Grid architectural model defined in Chapter 2 through the following concrete layers:

1. *Monitoring Layer* represents the platform dependent part inherent to (almost) every tool implementation. The purpose of the monitoring layer is to provide support for online tool development. It consists of a set of light-weight sensors distributed across all the individual machines that

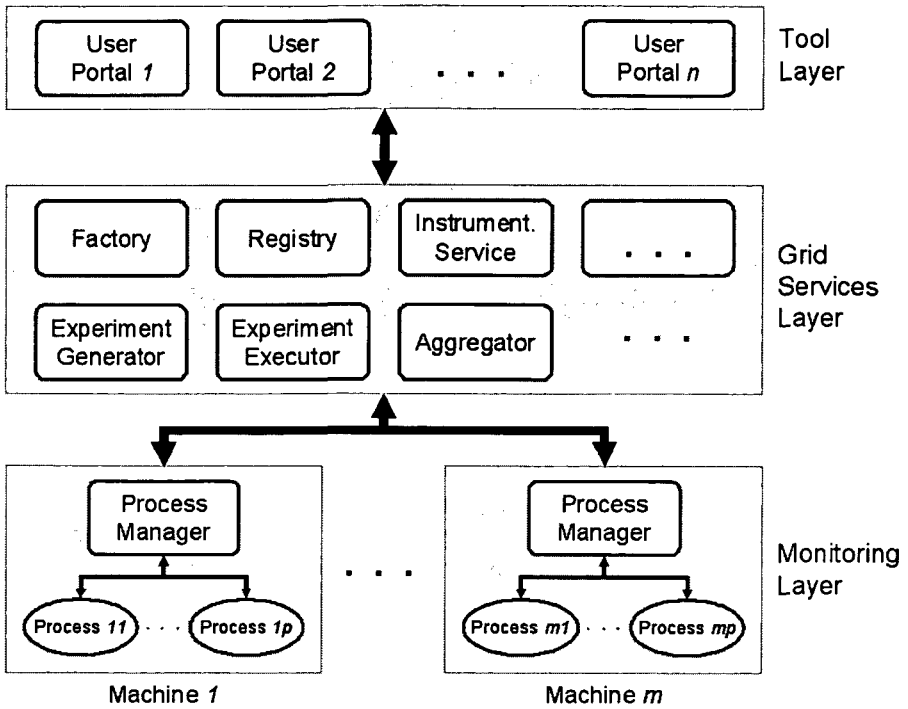


Fig. 5.1. The ZENTURIO tool integration framework architecture.

build in aggregation the Grid machine layer. The sensors typically extract and monitor low-level hardware and software features specific to every platform and operating system. The isolation of platform dependencies under a monitoring layer that exports a portable API reduces the effort of porting n tools onto m platforms from $n * m$ to $n + m$;

2. *Grid services layer* consists of an open set of high-level portable Grid services that can be dynamically deployed and instantiated on arbitrary Grid sites, as introduced in Section 2.7. The Grid services facilitate the tool development and enable the interoperability through the concurrent service use;
3. *Tool layer* consists of the end-user software tools, represented either by graphical user portals, or by simple batch front-end programs.

The functionality of a tool developed within this framework is no longer stored within a single monolithic front-end application acting as a black-box, as it has been traditionally done so far. Instead, the functionality is exposed and distributed amongst many small and reusable Grid services, often orchestrated in a loosely-coupled workflow. The tool interoperability is achieved by two design properties of this service-oriented architecture:

1. The Grid services can serve concurrent requests coming from potentially different remote clients (i.e., user portals representing potentially different end-user tools);
2. The monitoring sensors can simultaneously be called by multiple Grid services. This allows multiple clients concurrently monitor and manipulate the same physical processes and target machines.

Another important objective of the framework proposed in this chapter is to provide an extensible architecture open to further integrations and developments. Extensibility is related to the following three aspects in the proposed architecture:

1. *Add new services to the environment.* This translates to the ability to incorporate new Grid services and to add new tools to the framework. The Web services technology clearly separates the service interface specification from the service implementation which facilitates the addition of new services. The incorporation of new services requires the publication of their WSDL interface and (JAR file) implementation into the (UDDI) Service Repository, as specified in Section 5.3.2. The service implementation must allow multiple clients concurrently access and invoke operations with no knowledge of their mutual existence, which enables new client tools be naturally integrated on top of the existing Grid services;
2. *Extend existing components with new functionality.* Extending existing services with specialised versions through *delegation* is not only supported but encouraged by the framework. Since the Web services technology does not adhere to the object-oriented design principles, the extensibility through inheritance is not possible at the WSDL service interface level. Extensibility through *inheritance* is, however, possible at the Java class service implementation level;
3. *Implement new tools based on the existing services.* The tools implemented within the framework will interoperate indirectly through the common service use, as will be described in Section 5.7.1.

5.2 The Monitoring Layer

The Monitoring Layer consists of an open set of sensors that run on the target Grid machines and provide low-level information about the application processes and the system resources required for online tool development. The sensors can be remotely accessed through a portable platform-independent API developed on top of the light-weight Globus I/O library [66] and the Grid Security Infrastructure introduced in Section 2.4.

The design of the monitoring layer has been motivated by the following limitations of the SCALEA compiler-based instrumentation engine used by the ZENTURIO experiment management tool:

1. the compile-time instrumentation can be applied only once, prior to the application execution;
2. the application needs special preparation through specific compilation and link library options;
3. the performance analysis is done post-mortem based on the data stored in an Experiment Data Repository;
4. in order to interoperate, all the performance analysis tools will largely need to base their instrumentation run-time system on SCALEA.

The remainder of this section presents a general purpose instrumentation and monitoring sensor called Process Manager which aims to complement these limitations by using the dynamic instrumentation technology.

5.2.1 Dynamic Instrumentation

Dynamic instrumentation is a non-conventional instrumentation technology based on the insertion of binary code snippets at run-time into an already executing program. Dynamic instrumentation has several unique characteristics that make it suited for tool interoperability since it does not conflict with other existing instrumentation technologies:

1. it requires no advanced preparation of the application program, like special compilation options or link libraries;
2. it allows the instrumentation of binary programs compiled from any programming language, even of proprietary applications for which the source code is not available;
3. the instrumentation snippets can be inserted and removed from the code at any time which keeps the intrusion minimum.

The *Dyninst* [24] C++ library provides a machine independent interface for run-time code patching using the (platform-dependent) dynamic instrumentation technology.

Figure 5.2 illustrates the basic mechanism used by the dynamic instrumentation to insert code snippets into a running process. The machine instruction code is inserted into the process by replacing an instruction located at the desired instrumentation point with a branch to a code snippet called *base trampoline*. The base trampoline saves and restores the process state before and after executing the instrumentation code. The specific instrumentation code is contained within a *mini trampoline* that can be inserted either before or after the relocated instruction.

The main limitation of the dynamic instrumentation is the exclusive focus on binary executable object code. While the compilation for debugging purposes (i.e., usually the `-g` compilation flag) produces binary code that largely matches the source code, the compilation for high-performance execution usually generates a highly optimised executable that can no longer be uncompiled to the original source code. The limitation becomes even more

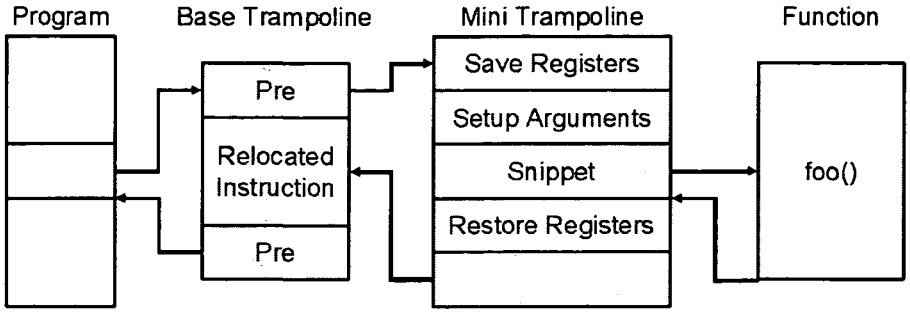


Fig. 5.2. The dynamic instrumentation control flow.

critical for high-level parallel programming languages like HPF and OpenMP, for which the dynamic instrumentation cannot be used for computing high-level language metrics associated with specific language directives. Moreover, porting the Dyninst library on different operating systems (and even upgrading it to different system and compiler versions) is a hard challenging task that critically impacts the implementation reliability and availability.

The source code instrumentation performed by SCALEA remains therefore of important value that is kept within ZENTURIO along side the dynamic instrumentation. The framework is open to the integration of further sensors like those provided by the SCALEA-G [162] Grid performance tool.

5.2.2 The Process Manager

The *Process Manager* sensor is a light-weight daemon (implemented in C++) in charge of controlling and instrumenting running application processes on a single machine. The Process Manager provides two mechanisms for connecting to an application process required to perform dynamic instrumentation:

1. *create* a process by providing the complete execution command and the input arguments;
2. *attach* to an existing process by providing the operating system process identifier.

The Process Manager serves instrumentation requests coming from remote Grid services, in particular from the Dynamic Instrumentor service which will be described in Section 5.3.7. Typically the Process Manager sensors do not communicate with each other, however, there may be special cases when such interaction is required (see Section 5.2.3).

The functionality offered by the Process Manager can be classified into five categories which are implemented by four threads as shown in Figure 5.3 and described in the reminder of this section.

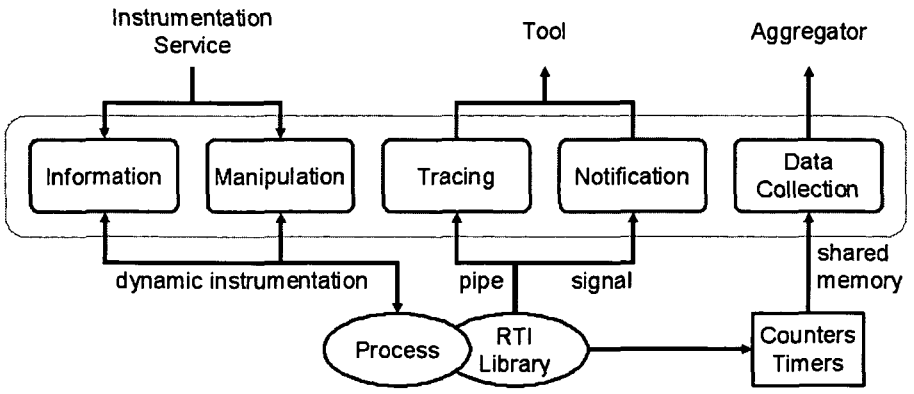


Fig. 5.3. The Process Manager architecture.

Information Functions

The *information functions* provide structural information collected from the running application processes. This includes the object code structure of each process and (global) variable values. The information is extracted from the binary executables and is complementary to the source code information, if available. Since retrieving the object code structure is a rather intrusive operation to be repeatedly invoked, the Process Manager extracts and caches the entire object structure through one single call after attaching to a running application. The cache is refreshed whenever a running process issues a UNIX system call `exec` that overwrites the entire process images, or when a dynamic shared library is loaded.

The object code structure of a process largely matches the original source code in the case when the application is compiled for debugging purposes (e.g., typically using the `-g` compiler option). In the case of highly optimised applications, however, the mapping from the binary executable to the source code becomes impossible due to complex irreversible compiler optimisation transformations.

Manipulation Functions

The *manipulation functions* are primarily used for dynamically injecting instrumentation probes into running application processes so that information about their execution may be gathered.

The *Run-Time Instrumentation Library* is a UNIX shared library [148] designed to ease the instrumentation of running application processes with high-level probes. The library is dynamically loaded by the Process Manager into the address space of each monitored process at run-time which enables the instrumentation of unmodified binary executables. The run-time instru-

mentation library provides the following probe types, hierarchically depicted in Figure 5.4:

1. *Timers* (i.e., wallclock, user, and system time) are associated with a set of start and stop instrumentation points;
2. *Counters* are inserted before or after any set of instrumentation points. There are two types of counter increments that can be provided:
   a) *constant* (e.g., usually one, in case of function call counters);
   b) *type size* used for counting the size of data structures (e.g., number of bytes passed as argument to various functions);
3. *Traces* are generated by inserting instrumentation probes that generate selective focused trace information;
4. *Notifications* insert probes that generate asynchronous events which are sent by the Notification thread to the subscribers using the push event model (see Section 5.4);
5. *Breakpoints* stop the application process whenever the program counter reaches a certain instrumentation point.

The Process Manager provides one instrumentation function for each probe type, which is responsible for generating and inserting the appropriate binary instrumentation snippets into the running process using Dyninst. Each instrumentation probe inserted into a running process is identified by a unique handler that can be used to *remove the probe* if it is no longer needed. Before instrumenting the application, the Process Manager checks whether the requested probe has been previously inserted and, if so, it returns the already allocated handler, thus avoiding instrumentation redundancy and minimising the intrusion.

Data Collection

The online performance data collected by the instrumentation probes is stored in a memory segment that is shared between the Process Manager and the application process. From this shared memory, the performance data is sampled by the *data collection* thread with minimum overhead and forwarded to the tool (or to an Aggregator service – see Section 5.3.8) for online performance analysis via an asynchronous notification callback. Each performance metric has its own associated online *sampling rate* which is specified as part of the instrumentation request.

Tracing

The *tracing* thread collects selective trace information generated by the application trace probes associated with certain instrumentation points. To simplify the implementation, a simple trace data format has been used that is sufficient for the development of the prototype tools that validate the framework (see Section 5.6). The trace data contains the following fields, which are currently restricted to the function level instrumentation granularity:

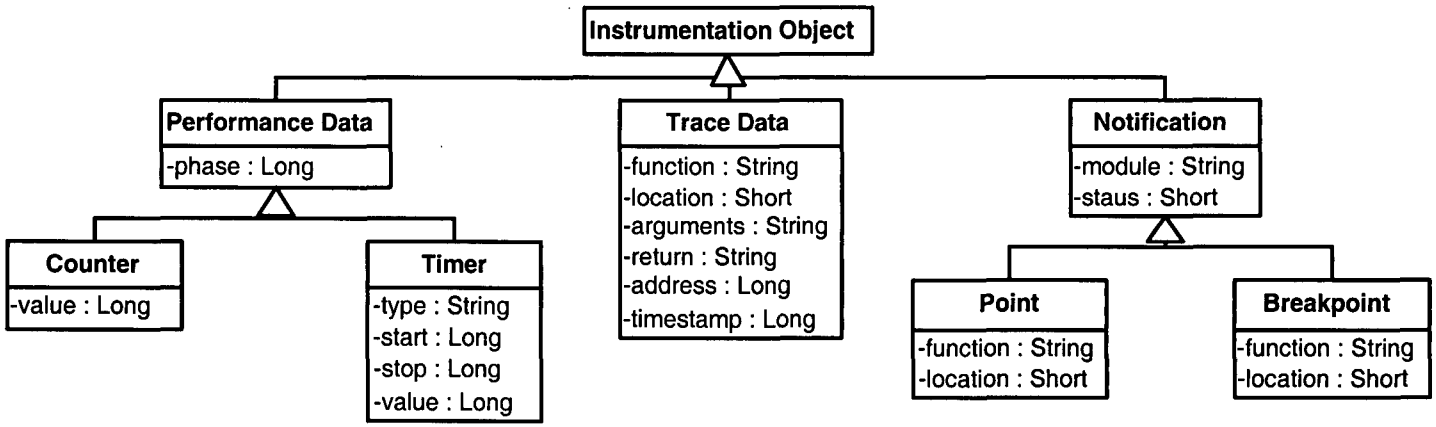


Fig. 5.4. The instrumentation probe class hierarchy.

1. *function name* in which the instrumentation point is located;
2. *location* of the instrumentation point (function entry, exit, or call);
3. *argument list* to the function (if a function entry or a call point);
4. *return* value of the function (if a function exit or a call point);
5. *address* of the instrumentation point (needed to distinguish between different calls to the same function);
6. *timestamp* when the trace has been generated.

As the trace data could get infeasibly large to be stored in the shared memory, it is periodically appended to a FIFO (First In First Out) file (or pipe) [148] from where it is (albeit less efficiently than the performance data) collected by the Process Manager.

Notification

The *Notification* is a light-weight sleeping thread that is awaken through UNIX signals [148] by the notification probes when certain events happen during the process execution. As the signals can only achieve process synchronisation (not communication), the information that describes the occurring events is stored in a special data structure within the shared memory segment (i.e., between the Process Manager and the executing process). The monitored application process and the Process Manager synchronise their access to this data structure by means of a UNIX semaphore [148]. This additional synchronisation is required since multiple simultaneous events may overwrite the data structure or exhaust the shared memory segment before the asynchronous notification thread manages to consume and forward the events to the requesting tool. There are three types of notifications handled by the framework:

1. Arrival at the instrumentation point;
2. Load or unload of a shared library by trapping the `dlopen` UNIX system call [148]. This notification is used by browsing tools to provide an updated view of the application object code structure (see Section 5.6).
3. Status change (e.g., *started, stopped, running, terminated*) that allows the tools to dynamically monitor and react upon any modification in the application status. The *stopped* state is usually caused by a correctness debugger and augments the experiment state transition diagram presented in Figure 4.5 (see Chapter 4).

5.2.3 Dynamic Instrumentation of MPI Applications

The Process Manager sensor has been designed for dynamic instrumentation of generic processes, with no particular focus on any programming paradigm. The use of higher level parallel programming paradigms, however, require extensions to the existing functionality. This section presents a specialisation (through C++ inheritance) of the Process Manager sensor to support MPI parallel applications.

The challenge in creating an MPI application for dynamic instrumentation is the need to obtain the identifiers of all the MPI processes, which have to be created through the Process Manager on each individual machine. Although MPI provides a standard interface of communication between parallel processes, it does not standardise the mechanism in which the parallel applications are created [145]. Currently each MPI implementation provides its own customised flavour of the `mpirun` command which starts an SPMD (Single Program Multiple Data) program across the nodes of the parallel machine [1]. The MPI-2 [116] specification aims for a standardisation of the `mpirun` command which is named `mpiexec`, unfortunately it contains only advises rather than a full portable script to be adopted by all MPI implementations. The MPI Forum argues that the range of the environments is so diverse (e.g., there may not even be a command line interface to invoke `mpiexec`) that MPI cannot mandate such a universal mechanism.

Since a universal MPI application start-up is not possible, this section chooses the widely spread MPICH [79] implementation for a case study. The technical scenario of creating an MPICH application for dynamic instrumentation is depicted in Figure 5.5. The client (i.e., the Dynamic Instrumentor service described in see Section 5.3.7) requests that the Process Manager create an MPI application by invoking the (MPICH specific) `mpirun` command. The Process Manager appends the `-t` execution flag to the `mpirun` arguments that executes the command in the test mode. The result returned by the `mpirun` test command represents the list of machines (i.e., processors of the parallel machine) where the MPI processes will be started. The last entry in this list is the *master process* that has to be executed by the Process Manager on the local machine which will subsequently spawn the remaining MPI *slave processes*. The Process Manager appends the `-p4norem` flag when executing the master command, thereby preventing the master process from starting the slave processes automatically. Instead, the master process returns to the Process Manager the command required to manually start the slave processes on different machines. The Process Manager delegates this task to its counterpart running on the same machine where the slave has to be started. This is the only situation when direct communication between Process Managers is required.

After being created, all the MPI processes must be resumed so that the slaves can acknowledge their creation to the master within the `MPI_Init` function. As most of the tools require that the application be halted immediately after its creation, a breakpoint is inserted at the end of the `MPI_Init` function of each process. A call to `PMPI_Comm_rank` is dynamically inserted before this breakpoint to retrieve the MPI process identifier within the `MPI_COMM_WORLD` communicator.

[1] For Multiple Program Multiple Data (MPMD) applications, the use of the standard library call `MPI_COMM_SPAWN` defined by the MPI-2 [116] standard solves the problem in a portable manner.

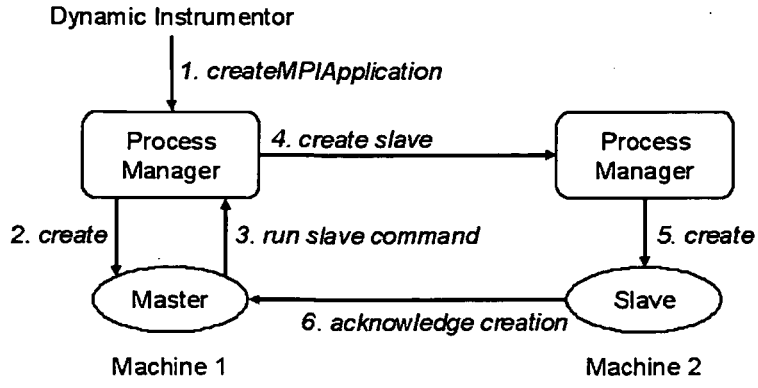


Fig. 5.5. Starting an MPI(CH) application for dynamic instrumentation.

The implementation of this start-up mechanism raises an interesting I/O buffering problem for which the dynamic instrumentation as a general run-time code patching approach enables a very interesting and effective solution. When given the `-p4norem` flag, the `MPI_Init` implementation of MPICH uses the C language `printf` command to write the master output that indicates how to start the slave processes (see Figure 5.5). Since the standard output of the master is redirected to a FIFO file (or pipe) [148] by the Process Manager, no output will be received until the output buffer is flushed. Rather than modifying the MPICH source code (which for other proprietary MPI implementations may not even be available) to explicitly flush the buffer after the offending `printf` and rebuild the whole MPICH library (thereby forcing the use of a customised library version), the Process Manager forces the flush at run-time by dynamically inserting a call to `fflush(stdout)` on-the-fly using Dyninst. This enables the implementation to work on an original and unmodified MPICH library.

Dynamic instrumentation of the new MPI-2 `MPI_Comm_spawn_multiple` and `MPI_Comm_spawn` routines, required by the Multiple Program Multiple Data (MPMD) programming model, allow newly spawned MPI processes be discovered at run-time and instrumented.

The dynamic instrumentation technology enables the framework to profile MPI library calls with ease. The generic profiling and tracing operations of the Process Manager can be easily focused on the MPI library routines. The profiling interface defined by the MPI standard is of no benefit to ZENTURIO. It is sufficient to apply the profiling and tracing operations to the `PMPI_`-prefixed calls directly, without using the `MPI_`-prefixed wrappers. Furthermore, apart from the MPI application start-up which unfortunately is not fully standardised, all the metrics and tools developed can be applied on any (even proprietary) MPI implementation.

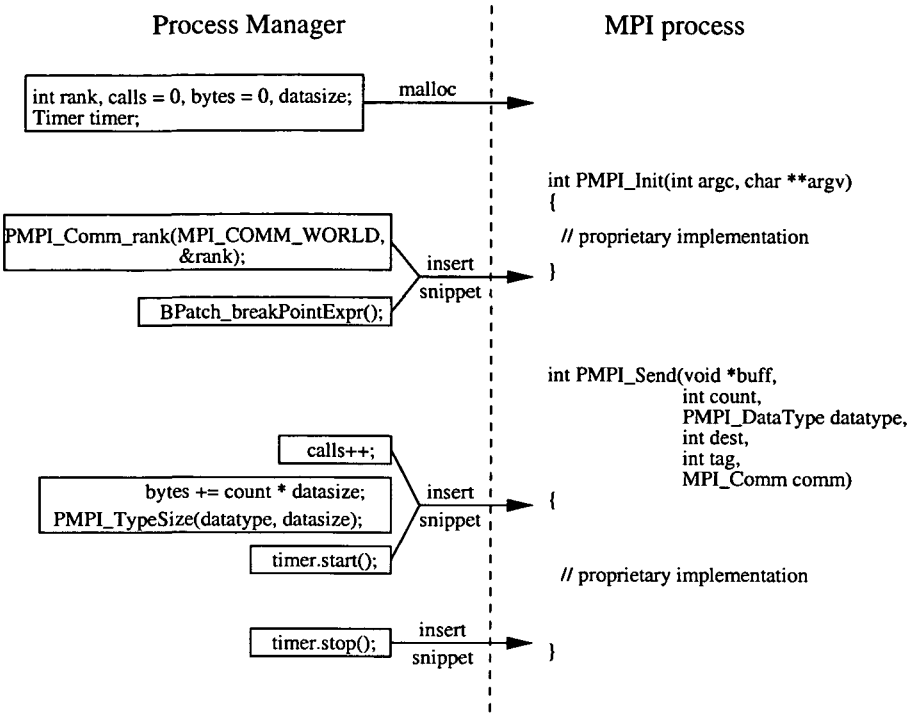


Fig. 5.6. Dynamic MPI library profiling.

5.3 The Grid Services Layer

The design and development of the middle Grid services layer within the tool integration architecture depicted in Figure 5.1 pursues the following goals:

1. the services provide a broad high-level and platform independent functionality required for tool development;
2. the services can be accessed concurrently and independently by multiple clients which is essential for tool interoperability;
3. the services can be easily instantiated on arbitrary remote Grid sites required for efficient deployment on the Grid;
4. there are flexible and efficient means for discovering the services.

The Grid community has acknowledged the Web services as the de-facto ground technology for building service-oriented Grid architectures [62]. The Web services, however, only mandate the use of XML documents for expressing interfaces and interactions between stateless Web services. In contrast, Grid services that model stateful Grid resources require enhancements to the basic Web services technology with functionality regarding state data (including lifecycle) and asynchronous notifications, as introduced in Section 2.7.

Despite the extensive efforts in the research and industry arena, there is still no widely accepted standard for modelling stateful Grid resources using the Web services technology. The OGSI [163] standard proposed by the Global Grid Forum has failed to be acknowledged by the Web services community due to the object-oriented approach of modelling Grid services based on inheritance, lifecycle encapsulation, and service state as WSDL elements, that were not in-line with the stateless Web services principles. The delegation approach addressed by the WSRF [67] has been recently been proposed within the Organisation for the Advancement of Structured Information Standards (OASIS) [119], with a final specification and a compliant implementation yet to be realised.

The thesis exploits this transitory period as an opportunity to provide its own contribution through the specification and implementation of several original Web services extensions for the Grid within the Global Grid Forum [30].

5.3.1 Web Application and Services Platform (WASP)

The Web Application and Services Platform (WASP) [153] from Systinet is the Web services toolkit used to implement the Grid services layer in ZENTURIO, since it proved to be the fastest, the most robust, and the most easy to use product from a range of other implementations (including Apache Axis [70], Glue [51], IBM's Web Services Toolkit (WSTK) [59], and Sun's Web Services Developer Pack (WSDP) [114]) which have been evaluated in the year 2001.

The WASP Web services runtime environment for Java is compliant with the Web services model described in Section 2.2. The WSDL interface of each Web service is automatically generated using WASP-specific tools and is therefore implementation specific. Every Web service is designed and implemented by one Java class, deployed within, and executed by, the WASP hosting environment. Upon the deployment of a Web service, a Web service instance with an associated WSDL document are automatically generated by WASP. The general structure of a WSDL document has been presented in Section 2.2. Each automatically generated WSDL document of a Grid service deployed within the WASP hosting environment contains one service interface and one service instance section. The service interface has exactly one `portType` which is homonym to the Java class that implements the service. Each Java method is mapped to one `portType` operation. The service interface is represented by exactly one `service` element which contains one `port` that defines the URL address of the SOAP `portType` network protocol binding.

Figure 5.7 depicts the state transition diagram of a Grid service deployed within the WASP hosting environment. *Offline* is the initial state and indicates that the service is not in memory, but will be loaded by the Java RPC Provider (and transferred to the state *Enabled*) when a request arrives. In the *Active* state the service is processing one or more clients. The state *Stopping* indicates that a request to stop the service has been issued, but some

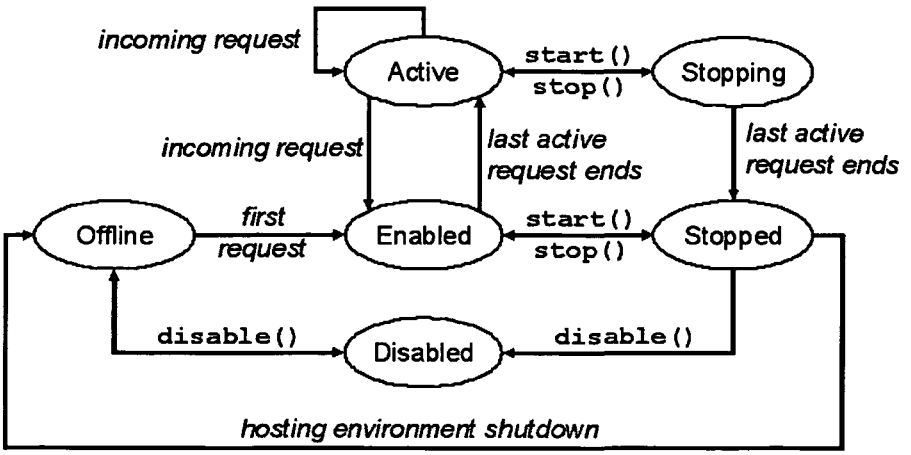


Fig. 5.7. The state transition diagram of the WASP-based Web services.

requests are still in process. A service in the state *Stopped* remains in memory but rejects all the incoming requests. *Disabled* means that the service is not in memory and cannot receive any requests. The transitions between the states are performed by the hosting environment either automatically (see the transitions marked with *italicised* text), or through explicit calls to the WASP administration service (see the transitions marked with `typewriter` style text).

5.3.2 Service Repository

The Universal Description, Discovery, and Integration (UDDI) [164] is a specification for distributed information registries of *persistent* business Web services.

One essential difference between business and Grid services, which in makes the use of UDDI in a Grid environment inappropriate, is the *service lifetime*. While a static UDDI registry (i.e., a database) is suitable for publishing information about static and persistent business Web services, it is certainly inappropriate for storing information about *dynamic* and *transient* Grid services.

In addition, publishing Grid services implementations in a Grid environment is crucial, as one cannot assume that the implementation code is available on the originally unknown remote site where the service instance would be desired. While compiled programming languages raise severe portability problems (especially due to unstandardised linkers and incompatible shared library dependencies), this issue is feasible for interpreted portable Java byte code in which the ZENTURIO Grid services are implemented.

This section proposes a slightly modified use of UDDI as a Service Repository for publishing *Grid service implementations* in a dynamic Grid envi-

ronment (transient Grid service instances are published within a specialised Registry service that will be described in Section 5.3.5). The UDDI model presented in Section 2.3.2 requires that the interface part of the WSDL document be published as a UDDI `tModel` and the instance part as a `businessService` element (i.e., as URLs). The `businessService` UDDI element is a descriptive container used to group related Web services. It contains one or more `bindingTemplate` elements which contain information for connecting and invoking a Web service. The `bindingTemplate` contains a pointer to a `tModel` element which describes meta-data of a Web service. An `accessPoint` element is set with the SOAP address of the service (`port`).

In contrast, the ZENTURIO Grid services use the UDDI `businessService` element to publish service implementation information of transient Grid service instances. The `accessPoint` element of a `bindingTemplate` is assigned the value of the URL to the JAR package that implements the Grid service.

The WSDL service interfaces and the service implementations are manually published by the users in the UDDI Service Repository. A notification mechanism compliant with the newest UDDI Version 3 specification can be used to inform the clients when new services are registered.

The Registry (see Section 5.3.5) and the Factory (see Section 5.3.4) are the only persistent services in ZENTURIO for which two entries corresponding to the service implementation and the existing (arbitrary in number) service instances are published in the UDDI repository. The distinction between the service implementation and a persistent service instance is made based on the `accessPoint` URL syntax. Persistent Factory instances have a standardised URL derived from the host name and a pre-defined port number (i.e., `http://hostname:port/Factory/`).

5.3.3 Abstract Grid Service

Figure 5.8 displays a hierarchical classification of the ZENTURIO Grid services, following the inheritance and state encapsulation model described in Section 2.7. Each service is a specialisation of the *Abstract Grid Service* that defines and partially implements the common functionality required by all the ZENTURIO Grid services. The Abstract Grid Service implements the `Producer` and `Consumer` interfaces that describe the *push events* of the generic event framework presented in Section 5.4.

The inheritance hierarchy is, however, only materialised at the Java implementation level, since the Web services technology does not adhere to object-oriented design principles. Each automatically generated WSDL document of a WASP-specific Grid service contains one single `portType` operation that merges the functionality of all the super-classes within the class hierarchy (see Section 5.3.1).

The Abstract Grid Service provides the following set of generic operations:

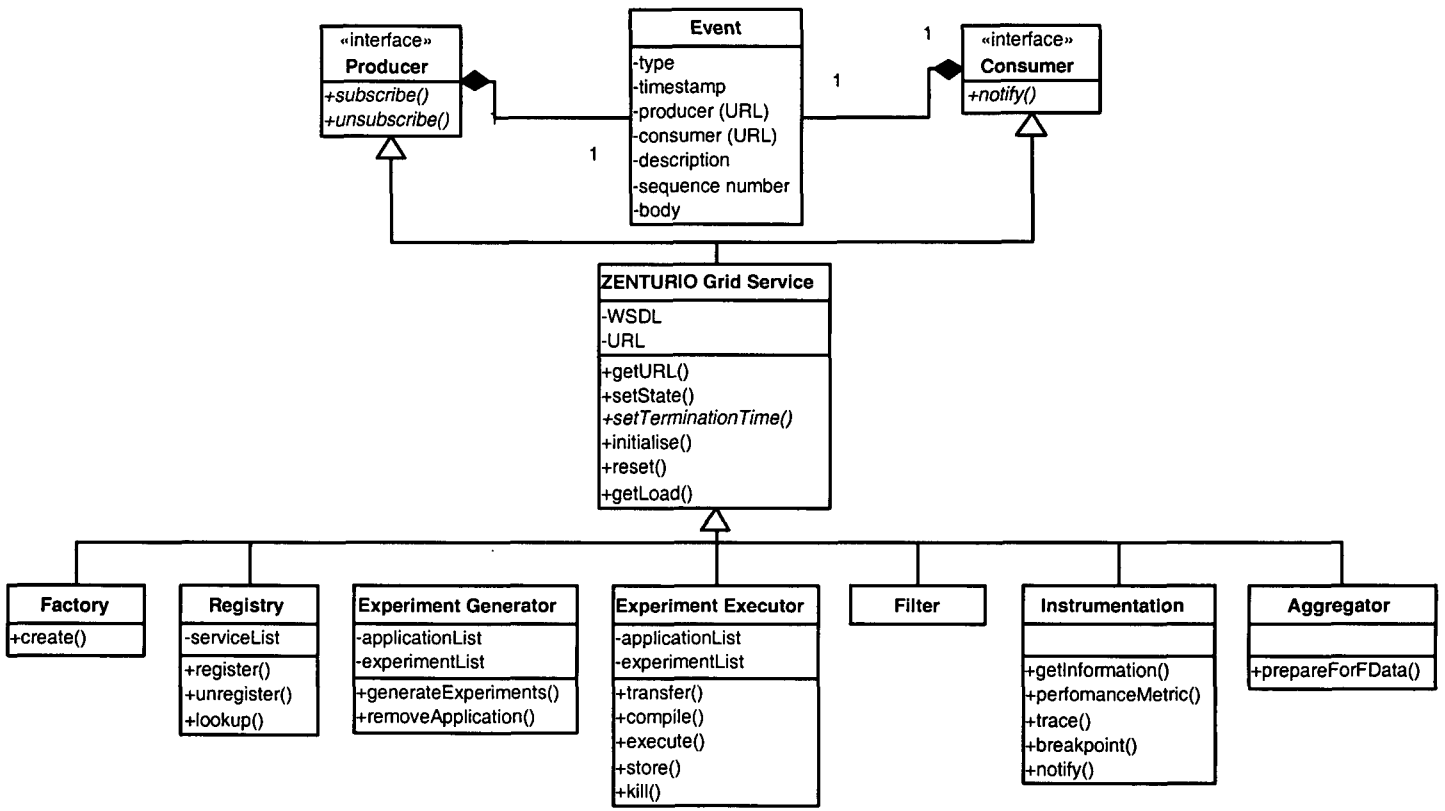


Fig. 5.8. The ZENTURIO Grid services hierarchy.

1. retrieve the URL of the WSDL file, constructed using the URL network location of the hosting environment plus a suffix path that uniquely identifies the service instance;
2. set and control the service state within the hosting environment;
3. retrieve and set the service soft-state termination time;
4. register the service with all the available Registries (retrieved from the UDDI Service Repository) and set the leasing time (see Section 5.3.5);
5. initialise the service after the transition from the state *Offline* to the state *Enabled*;
6. reset the service by eliminating all the state information when the service is changing to the *Disabled* state;
7. retrieve the load (in percentage) of a service.

The operations two and three implement the *lifecycle* of the ZENTURIO Grid services based on the WASP-specific API and the state transition diagram described in Section 5.3.1 and illustrated in Figure 5.7. The operations five to seven describe abstract *state* information and must be specialised by each ZENTURIO Grid service. Explicit service termination can be achieved by providing a termination time equal or prior to the current time. Destroying a Grid service requires to undeploy it from the hosting environment. A softer destroy method changes the service state to *Disabled* instead of undeploying it. A subsequent recreation of the service uses the existing disabled instance and changes its state to *Offline*, which avoids extra deployment and undeployment overhead.

The ZENTURIO Grid services are either persistent or transient. The Factory and the Registry are persistent services, while the others are transient. All the services can be *accessed concurrently* by multiple clients, which is an essential feature for interoperability in a Grid environment.

Each service method has a *synchronous* and an *asynchronous* version. The asynchronous version has the `Async` suffix and returns immediately an asynchronous receipt. Synchronous methods can be invoked against this receipt to check whether the asynchronous method has completed (optionally with a waiting timeout argument), or to get the return result, any input or output parameter, and any exception that may have been raised. This asynchronous method invocation style can be regarded as implementing the pull event model (see Section 5.4).

5.3.4 Factory

Each hosting environment that runs on every Grid site contains by default one persistent *Factory* service, which implements the *factory* abstract concept or pattern. The Factory is a generic service that creates and deploys (Java) Grid services of any type, which are previously packaged as JAR files. The Factory searches in the (UDDI) Service Repository for a service of a given type (i.e., as a `businessService` name – see Section 2.3.2). If such a service is found, the Factory creates a Grid service instance through the following steps:

1. get the URL of the service implementation (represented as an `accessPoint` element – see Section 5.3.2);
2. download the corresponding JAR package;
3. deploy the service in the same hosting environment in which the Factory resides;
4. initialise the service instance;
5. register the instance with all the Registry services (retrieved from the UDDI Service Repository);
6. set a leasing time equal to the service termination time;
7. return the URL to the WSDL file of the service instance.

The clients use this URL to retrieve the WSDL file and dynamically bind to the service through build-time generated proxies. Before searching for a service, the Factory examines within the hosting environment whether an instance of the same type has been previously destroyed and disabled. If such an instance is found, the Factory changes its state to *Offline*, thus saving expensive download, package, and deployment overhead (see Figure 5.7).

5.3.5 Registry

As opposed to other distributed service technologies (e.g., Jini lookup service [50] or CORBA Naming and Trading services [107]), the Web services do not provide any standard network-aware means of locating transient services (the limitations of UDDI have already been emphasised in Section 5.3.2). The Web services architecture [172] introduces the concept of *Discovery Agent*, but leaves its design and implementation unspecified.

The *Registry* is a persistent service which maintains an updated list of URLs to the WSDL files of the registered Grid service instances. The service URLs are organised in special purpose hashing tables for fast high-throughput service discovery. There may be an arbitrary number of persistent Registries residing on any Grid site which must be registered within the UDDI Service Repository. The Registry grants *leases* to the registered services similar to the Jini built-in leasing mechanism. If a service does not renew its lease before the lease expires, the Registry deletes the service from its internal service list. This is an efficient mechanism to cope with dynamic transient services and network failures. A leasing time of zero seconds explicitly unregisters the service. An event mechanism informs the clients (e.g., the user tools) about new Grid services that registered with the Registry, or when the lease of existing services has expired. Thereby, the clients are always provided with a dynamically updated view of the Grid services environment. The Registry is a generic service that operates on Abstract Grid Services and, therefore, can be used to register and discover services of any type within ZENTURIO.

The *Web Services Inspection Language (WSIL)* [10] defines a distributed Web services discovery method which is complementary to the UDDI centralised approach. WSIL defines an XML document that contains URL references to existing Web service instances (i.e., instance WSDL documents).

Each Registry service generates upon request one similar WSIL document which contains references to the registered transient Grid service instances. The WSIL document receives an associated creation timestamp that determines the validity of the data.

The Registry provides three types of methods for performing lookup operations:

1. *White pages* provide service discovery based on the service URL;
2. *Yellow pages* support service discovery based on the service type, compared against the (unique in WASP) `portType` of each service. As described in Section 5.3.1, the WSDL document of each WASP-deployed Grid service contains one single `portType` with the same name as the Java class that implements the service;
3. *Green pages* perform discovery based on the service functionality using the compatibility operator between two WSDL interfaces described in Section 5.3.6.

5.3.6 WSDL Compatibility

Functionality-based service discovery is a key feature in a Grid environment for which the Web services technology does not provide any standard support. An instance WSDL document $\mathcal{W}_1$ is defined to be *compatible* with $\mathcal{W}_2$ (denoted as $\mathcal{W}_1 \supset \mathcal{W}_2$) iff:

1. the set of `portType` names[2] of $\mathcal{W}_1$ instantiated by the `service` element is a superset of the corresponding set of $\mathcal{W}_2$;
2. for each `portType` of $\mathcal{W}_2$ instantiated by the `service` element, the set of `operation` names is a subset of the corresponding set of $\mathcal{W}_1$;
3. two operations with the same name are identical (i.e., have identical `parameterOrder`, input, output and fault messages).

The Web services compatibility operator is reflexive, antisymmetric, and transitive.

5.3.7 Dynamic Instrumentor

The *Dynamic Instrumentor* is a Grid service for dynamic run-time instrumentation of running parallel applications based on the functionality of the Process Manager sensor described in Section 5.2.2, augmented to apply on a distributed multiple process basis. The Dynamic Instrumentor provides the following four categories of operations:

1. *Information Operations* are based on the Process Manager information functions which include the retrieval of the application object code or the inspection of variable values. Because it is an expensive operation at

[2] *qnames* in the WSDL specification and terminology [31].

the Process Manager level, the Dynamic Instrumentor service retrieves the object code only once during the lifetime of a process (i.e., when the process is created or attached) and caches it for serving further requests;

2. *Performance Metric Operations* are based on the Process Manager manipulation functions, but operate at a higher level of abstraction, e.g., count number of function calls, compute the execution time of a function, count the number of bytes passed in a function parameter. On top of the generic performance metrics, a specialised *MPI Dynamic Instrumentor* builds MPI-specific metrics which include:
   a) the number of messages sent;
   b) the number of I/O operations (based on the MPI-IO [156] standard);
   c) the time spent in communication (i.e., by timing the routines from the `MPI_Send` and `MPI_Recv` family);
   d) the time spent in I/O operations;
   e) the time spent in synchronisation (i.e., `MPI_Barrier`);
   f) the number of bytes sent and received in communication;
   g) the number of bytes involved in I/O operations;
3. *Function Trace Operations* request that the entry, the exit, and the call points of a user, a system, or a library function are logged;
4. *Notification Operations* request that the client (i.e., the tool) be notified (using the push event model – see Section 5.4) when certain events (e.g., instrumentation point reached, shared library loaded, process forked or exited) occur in the application;
5. *Breakpoint Operations* request the insertion of normal or conditional breakpoints. Since a breakpoint only stops the process when it is reached by the program counter, a typical use is in conjunction with a Notification probe (see Section 5.2.2) that informs the client where such an breakpoint event has occurred (rather than reporting only a process status change).

5.3.8 Aggregator

When dealing with parallel applications, frequently the first step in processing the collected (performance) data (by the Process Manager data collection thread) requires a reduction step for better data understanding. The *Aggregator* is a generic Grid service that takes large amounts of data and, through the use of a chosen aggregation function, reduces it to more manageable quantities. The Aggregator supports reduction over time or across processors using a variety of aggregation functions including *mean*, *total*, *variance*, *sum*, *max*, or *min*. A more specialised metric for parallel processing is the *load balance* defined as the ratio between the mean and the max value. A value of one indicates the perfect load balance and a value of zero indicates the worse case load balance.

The Dynamic Instrumentor provides an Aggregator service when requesting from several Process Managers to collect dynamic performance data from

a parallel application. The Aggregator specialises both the `Consumer` interface for receiving data from the Process Manager (upon subscription), as the `Producer` interface for sending data to the client tool. Both interfaces are part of the push event model described in Section 5.4.

5.4 Events

The *Grid Monitoring Architecture (GMA)* [157] that emphasises the need of application and resource monitoring through asynchronous event notifications has been widely acknowledged within the Grid community. ZENTURIO designs and implements a generic event framework which largely confirms to the GMA, but has a broader scope not limited to performance events.

Definition 5.1. *An* event *is a timestamped data structure generated by a sensor and sent by a producer to a consumer. An event* producer *is a Grid service that implements the* `Producer` *interface and uses sensors to generate events. An event* consumer *is a Grid service (usually a thread within the client application) that implements the* `Consumer` *interface (see Figure 5.8).*

Sensors can be stand-alone like the Process Manager or embedded inside producers. Up-to-date information about the existing producers and consumers is maintained by the Registry service.

5.4.1 Representation

The ZENTURIO event representation combines the two approaches proposed by the GMA [157] with some subtle modifications that make the specification clearer. An event consists of two parts:

1. The *event header* is the standard part of the event structure that comprises the following fields:
   a) *event type* is an identifier that refers to a category of events defined by an *event schema*;
   b) *timestamp* indicates when the event has been generated. If the events are buffered, the elements in the event body may contain additional timestamp information. The timestamp representation uses the GMA standard proposal [157];
   c) *event producer* (URL);
   d) *event consumer* (URL);
   e) *sequence number*;
   f) *expiration timestamp*;
2. The *event body* represents the effective information carried by the event. It consists of the following four fields, where the last three are optional:

a) *homogeneous container* of elements where every *element* refers to a single event. The structure of an element (i.e., the type) is defined by an event schema;
b) *element description* (textual);
c) *measurement unit*;
d) *accuracy.*

The ZENTURIO event architecture depicted in Figure 5.9 supports three types of interactions between producers and consumers, as specified by the GMA:

1. *public/subscribe (PS)* is a generalisation of the *push model* where the initiator can be either the producer or the consumer. The initiator searches in the Registry service for the other party (producer or consumer) and registers for (the production or the consumption of) events. The producer sends events to the consumers until the initiator unsubscribes. The consumers subscribe for events to the producers by specifying the following inputs:
   a) *event type* that uniquely identifies the category of events desired;
   b) *event consumer* (i.e., URL of the WSDL file) that specialises the `Consumer` interface which receives the asynchronous notifications;
   c) *event parameters* specify the properties (characteristics) of the events to be sent to the user (e.g., process identifier for which status events must be sent). The event parameters describe an event and are therefore included within the event schema (i.e., as data members);
   d) *filter* specifies under which conditions an event must be sent (e.g., minimum value for a CPU load event);
   e) *subscription expiration time*;

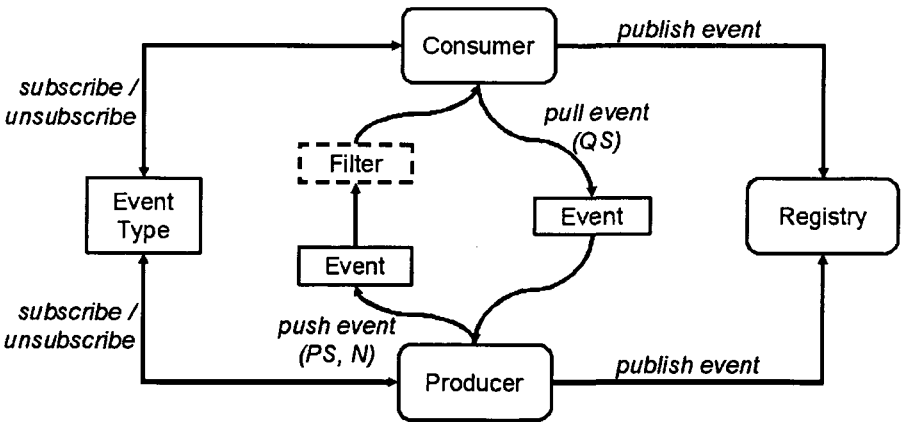


Fig. 5.9. The ZENTURIO event architecture.

2. *query/response (QR)* generalises the *pull model*. The initiator is the consumer and the event is sent in a single response any time after the event has been requested;
3. *notification (N)* is a slight specialisation of the push model. The producer transfers the events to the consumer in a single notification with no preliminary subscription.

Figure 5.10 depicts the generic event classification within ZENTURIO based on several event types. Table 5.1 displays the producers and the sensors for the event types supported by ZENTURIO. Table 5.2 gives a detailed description of the event types and their use within ZENTURIO.

Event Type	Producer	Sensor
Application Status	Experiment Executor Experiment Generator Dynamic Instrumentor	Experiment Executor Experiment Generator Process Manager
Process Status	Dynamic Instrumentor	Process Manager
Thread Status	Dynamic Instrumentor	Process Manager
Network Status	SCALEA-G	`netstat`
Site Status	SCALEA-G	`ping`
Service Status	Abstract Grid Service Registry	Abstract Grid Service Registry
Application Performance	Aggregator Dynamic Instrumentor	Process Manager SCALEA
Process Performance	Aggregator Dynamic Instrumentor	Process Manager SCALEA
Thread Performance	Aggregator Dynamic Instrumentor	Process Manager SCALEA
Network Performance	SCALEA-G	NWS
Site Performance	SCALEA-G	NWS
Service Performance	Abstract Grid Service	Abstract Grid Service

Table 5.1. The event implementation support in ZENTURIO.

5.4.2 Implementation

Events require support for *asynchronous messaging* which is not explicitly supported by the Web services specification. The most recent extended Web services architecture specification draft [172] mentions asynchronous messaging as an additional feature to be incorporated, while it is not clear who will provide a concrete specification (e.g., another W3C group, a vendor-specific implementation). The Web services standards, however, do include mechanisms on which asynchronous operations can be based. For instance, the Web services operations can be of type *one-way*, in which case no SOAP response

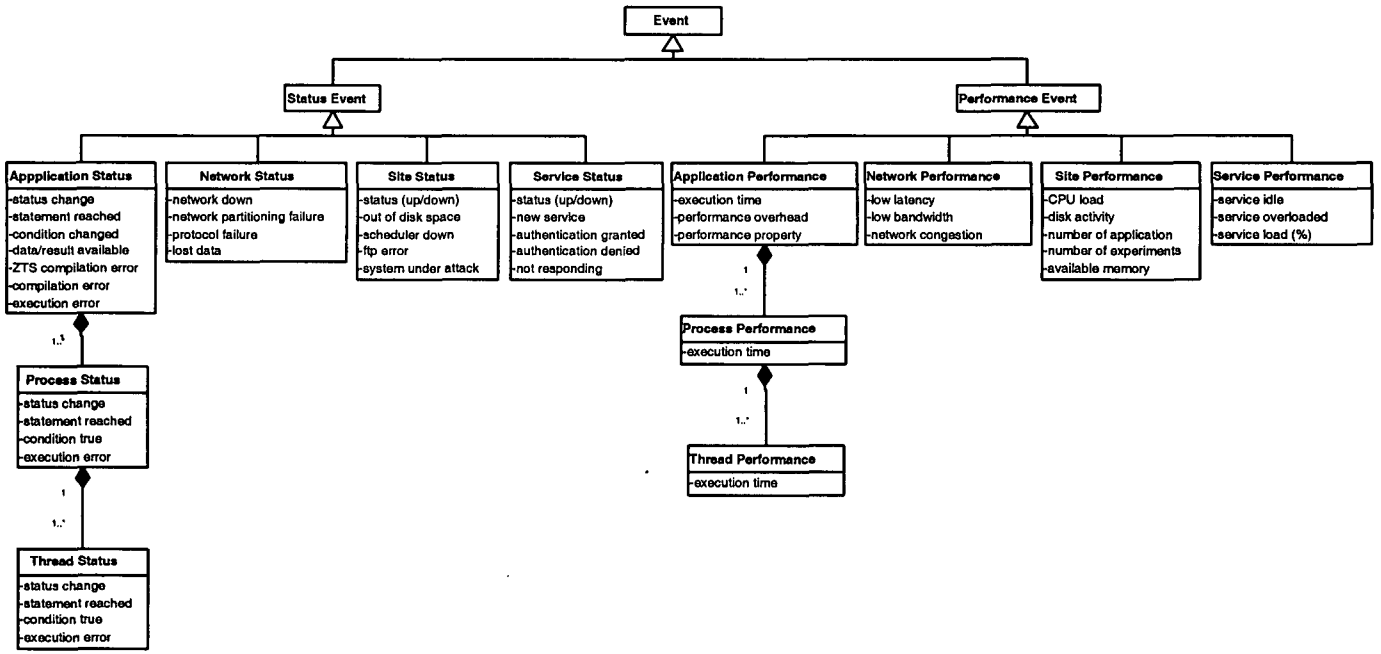


Fig. 5.10. The event hierarchy in ZENTURIO.

Event Type	Producer	Consumer	Sensor	Parameters	Filters	Event Elements	Interaction
service state	Registry	User Portal	Registry	type, site		status (up/down)	PS, QR
new service	Registry	User Portal	Registry	type site		service type, URL	PS, QR
authentication failed	Abstract Grid Service	User Portal	Abstract Grid Service			user name	N
out of disk space	Exp. Generator, Exp. Executor	User Portal	Exp. Generator, Exp. Executor			site	N
compilation error	Exp. Generator, Exp. Executor	User Portal	Exp. Generator, Exp. Executor			ZEN file, message	N
fie transfer error	Exp. Generator, Factory	User Portal	Exp. Generator			site	N
condition true	Exp. Executor	User Portal	Exp. Generator Instrum.	identifier, app.	bool-expr		PS
application performance	Dynamic Instrumentor	User Portal	Process Manager	metric, app., sample rate	bool-expr, max, min	value	PS
new experiment	Exp. Generator	User Portal	Exp. Generator	app., ZEN vars		experiment	PS, QR
experiment status	Exp. Generator, Exp. Executor	User Portal	Exp. Generator, Exp. Executor	app., ZEN vars		experiment status	PS, QR
scheduler down	Exp. Executor	User Portal	Exp. Executor			site, scheduler	N N
scheduler unsupported	Exp. Executor	User Portal	Exp. Executor			site, scheduler	N
service load	Exp. Generator, Exp. Executor	User Portal	Exp. Generator, Exp. Executor	site, sample rate	max/min load	no. apps, no. exps.	PS, QR
compilation error	Exp. Executor	User Portal	Exp. Executor			experiment, message	N
execution error	Exp. Executor	User Portal	Exp. Executor			experiment, message	N
Repository store error	Exp. Executor	User Portal	SCALEA			experiment, message	N
Repository access error	Exp. Generator, Exp. Executor	User Portal	Exp. Generator, Exp. Executor			site, message	N

Table 5.2. Overview of the supported ZENTURIO events.

is generated and only a HTTP notification is sent back. Moreover, WSDL defines operations of type *Notification* that allow an endpoint to send a message, however, it omits to define a network protocol binding for it (this will be solved in WSDL 1.2).

ZENTURIO implements the query/response events using the WASP-specific asynchronous methods with the `Async` suffix, as described in Section 5.3.3.

The implementation of the publish/subscribe and the notification interactions (both based on the push event model) are based on one-way Web services operations that reverse the roles of services and clients. The client takes the role of a service that receives one-way notification callbacks by implementing the `Consumer` WSDL interface. The implementation is based on WASP-specific *embedded servers* that allow one client to start a hosting environment as a separate thread. Although the declared purpose of this WASP feature is rapid prototyping, the embedded server enables synchronous or one-way callbacks. The SOAP address of the embedded server where the callbacks must be sent is given to the producer service during the event subscription.

The entire event implementation in ZENTURIO is, however, WASP-specific and therefore not portable across other Web services toolkits.

5.4.3 Filters

Filters can be either encoded inside the event producers (i.e., ZENTURIO Grid services) or designed separately, as special kind of intermediaries.

Definition 5.2. *An* intermediary *is a Grid service that insinuates between a producer and a consumer during a push (i.e., publish/subscribe and notification) event notification. A* filter *is an intermediary which delivers to the consumers a subset of the messages received from the producers.*

An intermediary can be shared by multiple producers and consumers. The event subscription method of the ZENTURIO producers can receive as input (along with event type, consumer, and event parameters) an array of filters of the Abstract Grid Service type. The filters are chained such that the first filter receives the messages directly from the producer and the last filter delivers the output messages to the consumer. This general method of chaining filters is employed at certain latency costs (unless the deviated path through the filter has a higher bandwidth than the direct path producer-consumer).

ZENTURIO implements an abstract template filter that specialises the Abstract Grid Service and has exactly one producer and one consumer. Filters can be easily plugged-in by specialising this abstract class and implementing the filtering algorithm.

5.5 Firewall Management

Firewalls are a critical topic in a Grid environment where geographically distributed Grid services need to transparently communicate through message exchange across multiple administrative domains. This section describes the pragmatic approach taken by ZENTURIO for traversing firewalls, however, a proper solution that satisfies all the security constraints is beyond its scope.

The Web services hosting environment described in Section 2.3.4 offers the advantage of a *single entry point* for accessing all the services through the provision of an embedded SOAP dispatcher. Each ZENTURIO Grid service has two associated communication *ports* that have to be remotely accessible:

1. *synchronous service port* is a property of the hosting environment and therefore is common to all the services hosted on a Grid site;
2. *asynchronous notification port* is a state property that must be exposed by each stateful Grid service.

All the hosting environments and the event consumers listening on open site ports are responsible for authenticating every request using the Grid Security Infrastructure mechanisms (see Section 2.4).

The ZENTURIO experience identified two serious obstacles in deploying large Grid infrastructures across different academic domains:

1. Independent (and in many cases not interacting) system administrators usually restrict the access to the open ports to certain trusted administrative domains. Various scenarios defined by the community within the Global Grid Forum, however, often require more flexibility. One frequently mentioned requirement of the Grid users is *mobility*, i.e., the ability to connect and use the Grid from arbitrary Internet locations (e.g., during conferences), possibly from unknown IP addresses (e.g., received through the Dynamic Host Configuration Protocol (DHCP) [142]) which are commonly rejected by any firewall;
2. To receive events at the client sites is usually impossible, the following two scenarios being often encountered:
   a) firewalls at foreign Internet sites outside the Grid infrastructure (e.g., where a demo is wanted) where it is impossible to ask the system administrators for any firewall changes;
   b) the use of the Network Address Translation [99] mechanism.

Table 5.3 displays the set of firewall ports that must be remotely accessible within the Grid environment of ZENTURIO. The ZENTURIO experience relieved that restricting the access to the set of open ports across n sites requires the tight interaction of C_n^2 pairs of system administrators which is not scalable in a large Grid environment.

Port Type	Port Value
GLOBUS_TCP_PORT_RANGE	40000 - 40100
GRAM Gatekeeper	2119
GridFTP	2811
MDS	2135
NWS Slapd	2112
NWS Nameserver	8090
NWS Memory	8050
NWS Sensor	8060
NWS Forecast	8070
SCALEA-G	40600 - 40625
Hosting Environment	8080
Experiment Data Repository	5432

Table 5.3. The open firewall ports in ZENTURIO.

5.6 The Tool Layer

The ZENTURIO experiment management tool is the principal end-user tool built within the tool integration framework presented in this chapter. The ZENTURIO user portal is a thin client developed on top of the high-level Grid services layer that makes use of the static instrumentation technology provided by the SCALEA [161] performance tool and wrapped by the Experiment Generator service. The tool interoperability is achieved through the post-mortem share of the data stored into the Experiment Data Repository.

This section describes a complementary set of interoperable prototype tools that use the dynamic instrumentation technology for online application analysis. The key feature of the tool-set is the run-time interoperability, achieved through the common use of the Dynamic Instrumentor and the Aggregator Grid services on top of the shared Process Manager sensors. The tools operate on unmodified executable files and can be used to monitor both user and system functions even when there is no source code available. The tools are generic and do not depend on any compilation options and linking libraries, flags, or any other preparation step.

5.6.1 Object Code Browser

The *Object Code Browser* is a graphical browsing tool which displays the object code structure of a given process retrieved from the application binary executable file. In the case of MPMD parallel applications, the union of the object structures of all the parallel processes is displayed. The Object Code Browser can be used in cooperation with the other tools for selecting the instrumentation focus (see Section 5.7.2). The Object Code Browser subscribes to the Process Manager for event notifications upon changes in the object structure of the application that require the following display updates:

1. *fork:* show the new process and its object structure;
2. *exec:* reload the modified process object-code;
3. *dlopen:* add the dynamic shared library to the list of application modules displayed;
4. *exit:* delete the process from the parallel application process list;
5. *status change:* update the process execution status.

5.6.2 Function Profiler (Z_prof)

The *Z_prof* function profiler, analogous to the UNIX tool `prof`, displays the call-graph profile data by timing and counting selected function calls. The MPI flavour of the tool offers functionality to:

1. *count:*
   a) the number of messages sent and received (from the `MPI_Send` and `MPI_Recv` family);
   b) the number of bytes sent and received;
   c) the number of I/O operations (based on the MPI-IO [156] specification);
   d) the number of bytes involved in I/O operations;
2. *time:*
   a) the communication routines;
   b) the synchronisation routines;
   c) the I/O routines.

Additionally, similarly to the UNIX administration tool `top`, the tool can be configured to display the first n functions in terms of the invocation or the execution times. The information is provided online, as the application executes. The refresh interval is determined by the input data sampling rate indicated during the dynamic instrumentation.

5.6.3 Function Tracer (Z_trace)

Z_trace is an online tool that traces in the style of the UNIX software tool `truss` the functions executed by an unmodified executable binary application. The tool does not differentiate between user, system, or library calls and does not require source code information. However, in order to be able to extract the function input and return arguments from the stack, the type information is required to be present in the binary executable. Therefore, the application needs to be compiled with appropriate flags (i.e., usually `-g`), otherwise only the function name is returned. To manually provide the function signature to the tracer is platform dependent and is not always a feasible solution. Since the object code and the function set of most programming languages is rather large and uninteresting (e.g., the smallest C++ program has about 1500 functions, most of them located in the `libc` library), it is recommended that the tool be focused on an interesting subset of functions or application modules. The

```
algorithm callgraph_tracer
  instrFunc = { }
  Z_trace("main")
end algorithm

procedure Z_trace(func)
  instrFunc = instrFunc + func
  tracePoint(func.entry)
  tracePoint(func.exit)
  for each callPoint in func.callPoints
    if(! callPoint.calee in instrFunc)
      addNotification(callPoint)
      addBreakpoint(callPoint)
    end if
  end for
end procedure

procedure Notify(callPoint)
  Z_trace(callPoint)
  removeBreakpoint(callPoint)
  resume()
end procedure
```

Fig. 5.11. The incremental callgraph tracing algorithm.

focus can be indicated either as an input configuration, or graphically using the Object Code Browser (see Section 5.7.2).

Since the application object code is rather large, it is impractical and inefficient to pre-instrument all the application points with trace probes before starting the execution. Rather, the functions are instrumented incrementally before being executed, as sketched by the *incremental callgraph tracing algorithm* in Figure 5.11.

1. `Z_trace` is the main function trace routine that inserts trace probes at the function entry and all the exit points which have not yet been instrumented. Additionally, it inserts notification probes at all the call points to trigger notification callbacks for each new function invocation that must be traced too. Since the instrumentation is performed while the application is running, each notification has to be combined with a breakpoint that stops the process allows the tracer to instrument the new function before executing it;
2. `Notify` is the callback triggered by the notification probes and the Process Manager on behalf of the first invocation of each function. As a consequence, the tracer instruments the new function with trace probes by calling the `Z_trace` routine, removes the breakpoint, and resumes the process.

```
algorithm callgraph_coverager
instrFuncs = { }
Z_cov("main")

procedure Z_cov(func)
  instrFuncs = instrFuncs + func
  addCounter(func.entry, rate)
  addCounter(func.exit, rate)
  for each callPoint in func.callPoints
    if(! callPoint.calee in instrFuncs)
      addNotification(callPoint)
      addBreakpoint(callPoint)
    endif
  endfor
end procedure

procedure Notify(callPoint)
  Z_cov(callPoint.calee)
  removeBreakpoint(callPoint)
  resume()
end procedure

procedure DataCol(counter)
  if(counter > 0)
    deleteCounter(counter)
    writer(counter.point has been hit)
  end if
end procedure
```

Fig. 5.12. The incremental callgraph coveraging algorithm.

5.6.4 Code Coverager (Z_cov)

The *Z_cov* tool imitates the UNIX tool `tcov` to produce a test coverage analysis on a function basis. The tool counts the number of times the program counter hits each instrumentation point. *Z_cov* is useful in practice for detecting dead code due to, e.g., redundant conditionals, or obsolete functions.

Similarly to *Z_trace*, *Z_cov* employs an *incremental callgraph code coveraging algorithm* sketched in Figure 5.12 that lazily instruments each function just-in-time before its first execution:

1. *Z_cov* is the main instrumentation routine that computes the coverage of one arbitrary function. Firstly, it inserts counters at the function entry and all the exit points. Similarly to the incremental tracing algorithm outlined in Figure 5.11, the coverager inserts notification probes at each call point, followed by a breakpoint that allows to instrument each function before executing it for the first time;

2. *Notify* is a callback from the Process Manager that trapped a call to a function that has not yet been instrumented. As a consequence, the coverager instruments the invoked function by calling the *Z_cov* routine, removes the breakpoint, and resumes the process;
3. *DataCol* is the callback routine from the Process Manager that contains the counting information. Each application point with a counter greater than zero has been hit by the program counter and requires no more instrumentation. The coverager therefore removes this instrumentation which reduces the intrusion in the running process.

5.6.5 Sequential Debugger (Z_debug)

Z_debug is a traditional sequential debugging server in the `dbx` or `gdb` style that provides the following functionality:

1. create and attach the process (i.e., operation required for dynamic instrumentation);
2. detach the process (i.e., disconnect and leave the process running);
3. manipulate the process state (i.e., stop, resume, terminate);
4. send a UNIX signal to the process;
5. read and write (global) variables;
6. insert and remove breakpoints at arbitrary instrumentation points;
7. insert and remove probes (i.e., counters, timers, traces, notifications) at arbitrary instrumentation points;
8. delete and replace function calls;
9. retrieve the object code information;
10. display and manipulate the process stack.

5.6.6 Memory Allocation Tool (Z_MAT)

Z_MAT is a memory allocation tool, inspired from Purify [82], that traces the C memory allocation functions from the `malloc` and `free` family (i.e., `malloc`, `realloc`, `calloc`, `memalloc`, `valloc`, `free`). The tool provides the following online functionality during the execution of the application:

1. display the memory allocation blocks;
2. display the totally allocated and the free heap size;
3. detect memory leaks (i.e., memory allocations with no corresponding free calls);
4. detect erroneous memory `free` calls that have no corresponding memory allocations (such bugs are often difficult to track and produce non-deterministic crashes);
5. display the amount of space allocated for the process data segment by instrumenting the `brk` and `sbrk` system calls.

The C++ `new` and `delete` memory allocation operators are compiled, e.g., by the `gcc` compiler to built-in functions (i.e., `__builtin_new` and `__builtin_delete`), which in turn call the `malloc` and the `free` memory allocation functions, followed by calls to the structure constructor, respectively destructor.

5.6.7 Resource Tracking Tool (Z_RT^2)

Z_RT^2 is a simple tool in the style of the UNIX `icps` that displays an online a list of the resources allocated by an running process by tracking several POSIX system calls:

1. `open / close` to display the open UNIX file descriptors;
2. `shmget / shmctl` to display the allocated UNIX shared memory segments;
3. `msgget / msgctl` to display the UNIX message queues;
4. `semget / semctl` to display the active UNIX semaphores;
5. `sigaction` to display the list of UNIX signals trapped by the process.

In addition, the tool displays a post-mortem list of warnings containing the set of resources which have been allocated and not freed by the process.

5.6.8 Deadlock Detector (Z_deadlock)

Z_deadlock is a tool that dynamically instruments the blocking MPI receive communication routines and checks for run-time inter-process communication cycles based on the message source process identifier.

5.7 Tool Interoperability

An important objective of the tool integration framework described in this chapter is the provision of an effective environment for tool interoperability. This section first classifies the various types of tool interaction, then illustrates several examples that demonstrate how synergy can be gained by interoperable use of tools.

5.7.1 Classification

The framework distinguishes between two types of tool interactions:

1. *Direct Interaction* assumes direct communication between the tools and is entirely determined by the tool design and implementation. This type of interaction happens exclusively within the tool layer and is independent of the underlying framework. For example, a performance tool may input the performance data to a steering tool that checks for a specific bottleneck, or a steering tool may directly ask a debugger to execute a command in order to optimise the program execution (see Section 5.7.3);

2. *Indirect Interaction* is a more advanced type of interaction that is transparently intermediated by the framework via the Grid services and requires no work or any particular knowledge from the tools. This scenario occurs in practice when the ZENTURIO Grid services interact with each other "behind the scenes" on behalf of the tools. The indirect tool interaction can be further classified as follows:
   a) *Co-existence* when multiple tools operate simultaneously on different parallel applications but share the same Grid service instances or sensors (i.e., utilise the same Process Monitor to instrument different application processes on the same machine);
   b) *Process Share* when multiple tools attach and instrument the same application process simultaneously. This type of interoperability has the potential of creating a variety of interesting interoperability scenarios, as described in the next sections;
   c) *Instrumentation Share* when the tools share instrumentation probes while monitoring the same application process in order to minimise their intrusion. This interoperation is automatically handled by the Process Manager sensor;
   d) *Resource Lock* when the tools require exclusive access to a specific resource. For example, a tool (through the user credentials – Section 2.4) can ask the Process Manager for a lock on a certain application resource (e.g., process, function) so that it may perform some accurate timing. The Process Manager allows no other user to instrument that resource, though the existing timers may be reused and sampled through the instrumentation share interoperability type.

Figure 5.13 shows a screen-shot of four interoperable tools (i.e., clockwise from top right: the Object Code Browser, *Z_trace*, *Z_cov*, and *Z_prof*) instrumenting and monitoring the same *Mandelbrot* MPI application instance. The tools are independently instrumenting and monitoring various functions within the same MPI process (i.e., host `cama`, pid 18462).

5.7.2 Interaction with a Browser

A common task for most run-time tools is to display the application resource hierarchy. This includes the application source or object code structure (i.e., modules, functions, and instrumentation points), machines, processes, and threads. Since it is implementation redundant that every tool independently provides this functionality, the responsibility can be given to a single tool like the Object Code Browser introduced in Section 5.6. Apart from displaying the resource hierarchy of an application, the Object Code Browser can also be used to specify which resources are to be used when another interoperable tool is started.

The advantage of this interoperability is that tools such as *Z_prof* never need to manipulate the list of application resources. By selecting a set of

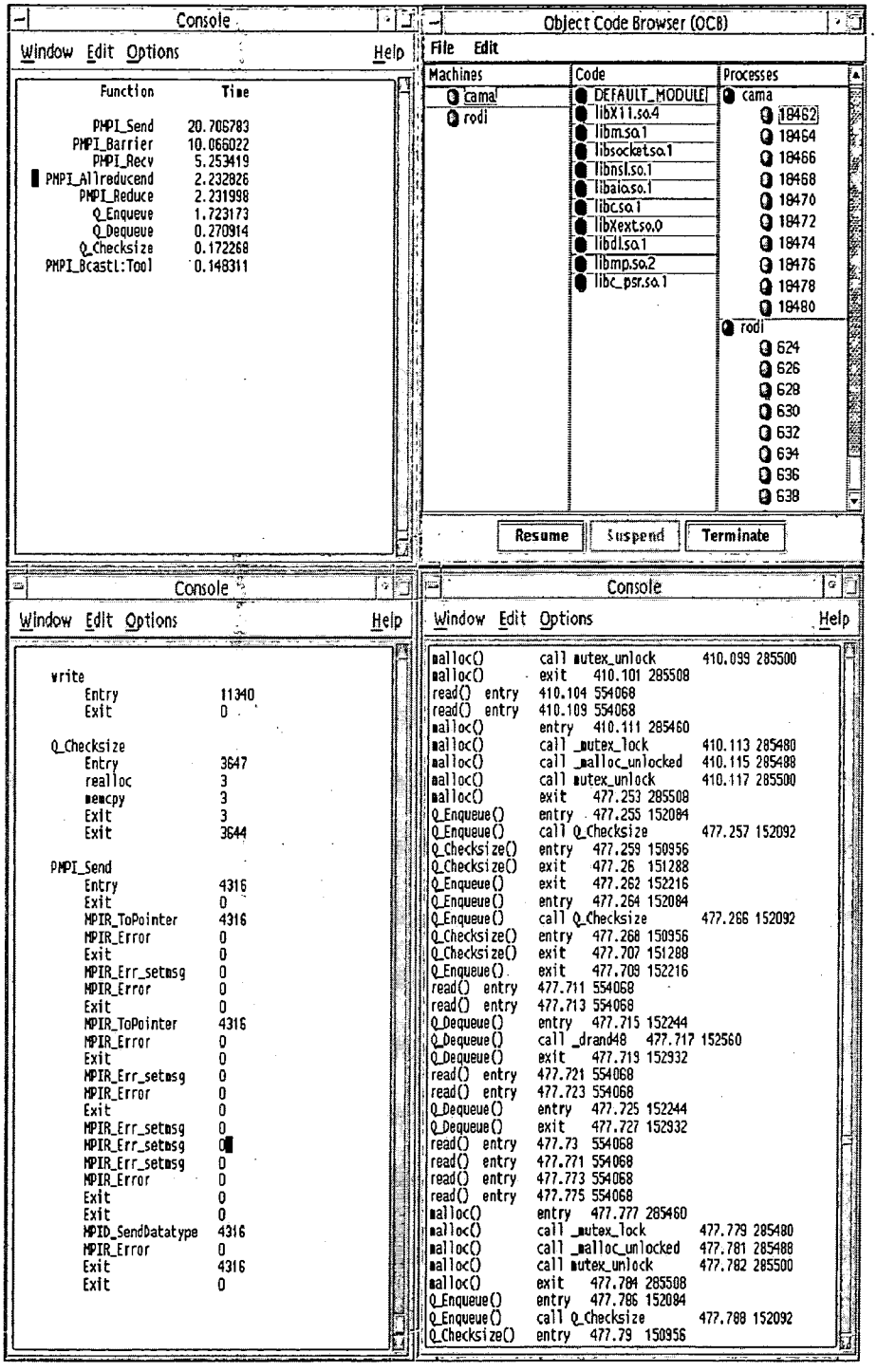


Fig. 5.13. A snapshot of the interoperable software tools.

functions in the Object Code Browser and running *Z_prof* with no other arguments, the selected functions will be automatically profiled.

5.7.3 Performance Steering

Performance optimisation is a non-trivial activity that typically consists of a four phase cyclic process [103] (see Figure 5.14):

1. *Performance Measurement and Data Collection* when a performance profiler is used to collect data from the application;
2. *Analysis and Visualisation* when performance analysis tools are used to interpret the performance data. Visualisation diagrams may be optionally employed if the analysis process is deferred to the end user;
3. *Optimisation* when the programmers choose various options to improve the performance of their programs. This is the main task of the performance steering tool;
4. *Modification* when the optimisations decisions taken at the previous step are applied to the program.

Once these four stages have been completed, the performance tool again evaluates the application performance and, if the result is still not satisfactory, the cycle repeats.

There are two options in which such a steering tool can be realised:

1. *static offline* targets the application optimisation through repeated execution for various parameter instantiations. This technique will be addressed in Chapter 6;
2. *run-time online* targets the application steering within one single execution. This scenario is approached in the proposed interoperability framework as follows:
   a) *The Performance Monitor* (e.g., *Z_prof*) collects the performance data and presents it in an appropriate manner to the steering tool. It might also highlight sources of performance bottlenecks;

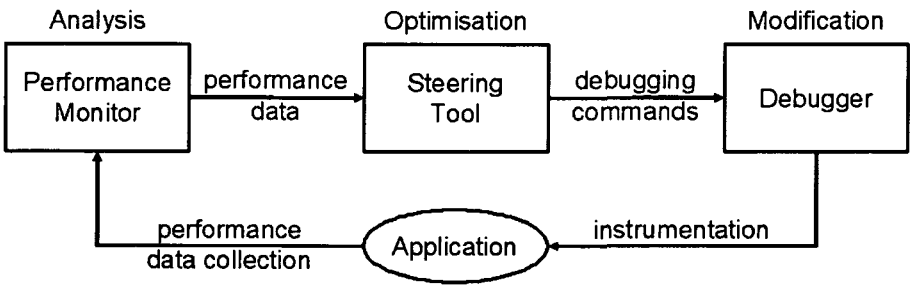


Fig. 5.14. The Steering configuration.

b) *The Steering Tool* decides whether an optimisation is required based on the performance information received from the performance monitor. If yes, it decides on the application modifications to be applied and gives the debugger the appropriate commands;
c) *The Debugger* (e.g., *Z_debug*) modifies the run-time binary code according to the commands received from the steering tool by inserting or removing binary instrumentation snippets, or by tuning online variable values using the dynamic instrumentation.

The run-time online performance steering can be of two types:

a) *Interactive* when the steering tool is replaced by the programmer who drives the execution of the performance profiler, visualises and analyses the performance data, takes optimisation decisions, and maps them into debugger commands;
b) *Automatic* in which case the steering tool gives hints about the possible performance problems and generates alternatives to optimise the program.

The use of the dynamic instrumentation enables the steering process to take place dynamically within one application execution without restarting the application every time a modification has been made. The interoperability type between the three tools is mixed. The steering tool interacts directly with the performance monitor and the debugger. The performance monitor and the debugger interact indirectly, by concurrently manipulating the same application process using the same Process Manager.

5.7.4 Just-in-time Debugging

Using a traditional low-level debugger to verify the correctness of a program requires to execute and repeatedly stop the program to inspect its state. If an incorrect program state is detected, all that is known is that a bug lies somewhere between the last inspection point and the current execution point (see Figure 5.15). For parallel programs the problem gets significantly magnified due to their non-deterministic nature that leads to hardly reproducible errors. Deterministic execution tools [137, 138], possibly in conjunction with a checkpointing tool [109, 146], may help in reproducing the error. This cyclic debugging method is, however, a time-consuming process since the problem has to be repeatedly reproduced. The real bottleneck is the fact that traditional instruction-level debuggers offer too low-level support for spotting erroneous program states and provide no information about their real cause. Furthermore, the deterministic reexecution tools used to reproduce erroneous program executions can be very time consuming for long program executions.

The *just-in-time* debugging concept attempts to eliminate the need of deterministically reexecuting the program by using of an online high-level bug detector to spot program defects in conjunction with a traditional low-level debugger to fix the problems on-the-fly using the dynamic instrumentation. Just-in-time debugging is an example of direct tool interaction.

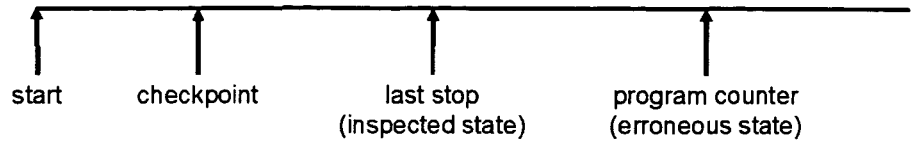


Fig. 5.15. The cyclic debugging states.

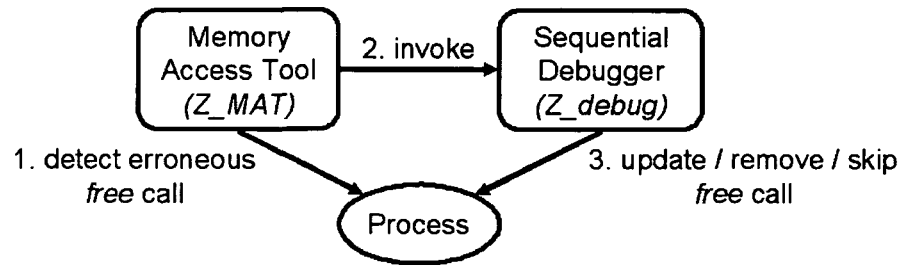


Fig. 5.16. A just-in-time debugging scenario.

In the proposed framework, the *Z_MAT* memory access tool can be used to detect memory access errors, like an attempt to deallocate a free memory location. In the just-in-time debugging configuration depicted in Figure 5.16, *Z_MAT* performs additional instrumentation that stops the application at the exact location where a memory access error is detected. Additionally, *Z_MAT* automatically invokes the *Z_debug* sequential debugger on the stopped process which gives the user the opportunity to analyse the problem at the exact location where it occurred and eventually pursue online corrections. In this example, changing the memory block pointer or skiping/deleting the `free` instruction are crucial for avoiding a highly probable crash.

5.7.5 Interaction with a Debugger

The interaction of software tools with a run-time interactive debugger requires special care since the debugger severely interferes with the process execution. The following two indirect interactions (i.e., process share) are of interest:

1. *Consistent Display* is an important task required by nearly any run-time tool. This issue becomes problematic when multiple tools are concurrently monitoring the same processes, since the display of each tool depends not only on its own activity, but also on the actions of other tools. When a visualiser like the Object Code Browser interoperates with a debugger, the following sample interactions are possible:
   a) if the debugger stops the program execution, the execution visualiser needs to update its display in order to show this fact;
   b) if the debugger changes the value of a variable, the (distributed array) visualiser must update its display with the new value, for consistency;

c) if the debugger loads a shared library in the application, or replaces a call to a function, the Object Code Browser must change its code hierarchy accordingly.

2. *Timing* is an important interaction can happen between a performance tool and a debugger. For example, *Z_debug* could choose to stop a process while the performance tool *Z_prof* is computing some timing operations. In this situation, while the user and the system times stop together with the process, the wallclock time keeps running. The framework takes care of this situation through the Process Manager that automatically subtracts from the wallclock counter the time during which the process has been stopped by the debugger.

5.8 WASP versus OGSI

The ZENTURIO Grid services infrastructure has been designed and implemented in the year 2001 based on the WASP toolkit, as described in Section 5.3.1. In the year 2003, the Global Grid Forum has finalised the Open Grid Services Infrastructure (OGSI) specification [163] that was aimed to be the standard technology for building Grid services. The extensions added by OGSI to the conventional Web services comprise standard means for managing the service lifetime (including time modelling), service data elements which expose service state within the WSDL `portTypes` of each service interface, and a standard interface for light-weight notification events. The Globus toolkit implements the OGSI specification within the Open Grid Services Architecture (OGSA) [62] based on the Apache Axis [70] SOAP implementation.

Within this wide international effort, the thesis brings its contribution by porting the ZENTURIO experiment management tool and the underlying Grid services to this new technology [126, 129]. The reminder of this section comparatively analyses various aspects that were substantially different in the WASP and the OGSI-based implementations, in particular issues regarding proxy management, service lifecycle, UDDI service repository, firewall management, Registry service, service throughput, and security.

5.8.1 Proxy Management

The ZENTURIO Grid services have been initially developed for the WASP server and SOAP engine for Java. The service deployment from the WASP to the OGSI-based implementation has been (in accordance with the Web services principles) straight forward, by using the corresponding automatic WSDL generation, packaging, and deployment tools.

Major difficulties have been encountered when porting the clients (i.e., the tools) which was mainly due to the different proxy management in the two SOAP implementations (which is not standardised by the Web services technologies). Interoperability between WASP-based clients and OGSI-based

services was also not a feasible solution, since the goal was to use and validate the OGSI extensions to the Web services (e.g., notifications, service data).

The proxy generation in WASP is dynamically generated at run-time during the service lookup and, therefore, completely transparent to the user. In contrast, the proxy generation for the OGSI Java clients is statically generated at compile time using a special `GSDL2Java` (`WSDL2Java` in vanilla Axis) tool. A limitation of this tool is that it generates not only stubs for transparent remote invocation of the services, but also Java Bean implementations compatible with the Axis `BeanSerializer` for all the complex types that appear as input or output arguments to the service methods. Each such bean contains the `set` and `get` methods to access the private data members, a default constructor, and additional bean (de)serialisation code. This means that the implementation of each complex type present in a service interface has to be a Java Bean, which is overwritten (or generated) by the stub generator.

This limitation is not imposed by WASP, which allows arbitrary non-trivial implementations of the complex types that are defined in the WSDL interfaces. The WASP serialisation is based on a `Reflection(De)Serializer` which manages the default type (de)serialisation using a Java Beans introspector that applies at run-time directly on the bean implementation class provided by the user.

The initial implementation of the ZENTURIO Grid services contained non-trivial implementations of several complex types to be (de)serialised (e.g., Experiment and ZEN-annotated Application classes). The code needed therefore to be redesigned, such that the stubs physically generated by Axis do not overwrite the original implementation and remove the non Java Bean methods. Two solutions have been considered to solve this problem:

1. ignore the Java Bean stubs generated for the complex types and paste the serialisation code into the implementation using a macro-processing tool (this method is simple but less neat);
2. reengineer the implementation in a class hierarchy, where the superclass is the Java Bean that will be overwritten by the stub generator, and the subclass contains the complex non Java Beans methods.

The second solution has been adopted which is neater, but requires a major reengineering of the application class hierarchy.

5.8.2 Service Lifecycle

One major contribution of OGSI to conventional persistent Web services is the standardisation lifetime management of transient Grid services (see Section 2.7). Normally each conventional Web service hosting environment implements its own state transition diagram which can be manipulated through specific API (see Section 2.3.4). The problem is that the implementation of transient services across various service containers is not portable.

WASP provides two different instantiation models of runtime published services and automatic lifetime management:

1. *Shared instantiation* is the usual instantiation method which shares one instance of the remote object across multiple clients. The service lifetime is controlled through WASP-specific TTL (Time-To-Live) routines as presented in Section 5.3.1 and Figure 5.7;
2. *Per-client instantiation* is a scheme through which the WASP hosting environment automatically creates a transient instance of a persistent service for each separate client on behalf of its first service invocation. This technique is similar with the WS-Context [25] standard for implementing stateful services.

This was an interesting occasion to notice that, while following a different development path than OGSI, existing advanced Web services toolkits like WASP do provide advanced proprietary extensions for implementing transient Grid services. The OGSI specification adds lifecycle as a property of Grid service instances, by defining a standard API as part of the `GridService` `portType` specification and by including termination time as a WSDL service data element. This solution has the key advantage of being portable across multiple OGSI-compliant implementations. Moreover, the service lifetime management is fully handled by the OGSI implementation toolkits which substantially simplifies the development of new transient Grid services.

5.8.3 UDDI-based Service Repository

Section 5.3.2 has presented a custom centralised repository for publishing persistent Grid services implementations based on the UDDI standard [164]. The generic WASP-based Factory service downloads the required service implementation from the UDDI Registry (if necessary) and deploys the service instance on-the-fly using the WASP runtime publishing tools. This run-time on-the-fly service deployment technique could not implement in the OGSI-based implementation that requires pre-deployment of persistent services before the hosting environment is started. Transient services are purposely designed for runtime deployment, however, the corresponding byte-code and WSDL interfaces need to be pre-deployed. This limitation is very critical in a Grid environment, where new services need to be deployed on new sites at run-time based on the dynamic resource availability.

5.8.4 Service Data

The OGSI most radical extension to the Web services is the ability to expose service instance state data for query, update, and change notification. The OGSI approach introduces a `serviceData` child element to the WSDL `portType` to describe stateful Grid services. Service data is an OGSI-specific feature and therefore not supported by WASP and any other traditional Web

services implementation. The ZENTURIO WASP-based implementation exposes the Grid services state through Java Bean `get` and `set` methods, which has the atomicity limitation exemplified in [163].

The service data elements exported by the OGSI-based ZENTURIO services are enumerated in Table 5.4. The service data elements of the Registry and the Factory services are implemented by the Globus toolkit as part of the VORegistry, respectively the `FactoryServiceSkeleton` implementation (the latter as an extension to OGSI).

Service	Service Data Elements
Experiment Generator	ZEN applications last experiment generated number of experiments Experiment Data Repository (JDBC URI) notification port
Experiment Executor	ZEN applications number of experiments (submitted/ queued/running/terminated/stored) Experiment Data Repository (JDBC URI) notification port
Registry	registered services notification port
Factory	created services UDDI URL notification port

Table 5.4. The ZENTURIO service data elements.

The service data elements were, however, one the major obstacles for the OGSI adoption within the Web services community due to their native object oriented roots that conflict with the stateless Web services principles.

5.8.5 Events

OGSI defines three different WSDL `portTypes` that aim to standardise the push event specification: `NotificationSource`, `NotificationSink`, and `NotificationSubscription`. Section 5.4 has presented the realisation of the push events in WASP based on embedded servers. This approach is also adopted by the Globus toolkit for the implementation of the OGSI `NotificationSink portType`. This was another interesting occasion to notice that existing Web services toolkits offer solutions to implement the OGSI extensions to Web services, although the declared objectives and development paths are different. In addition, OGSI specifies an event subscription mechanism on `serviceData` element changes, like those described in Table 5.4.

Support for the pull event model is standardised in OGSI by means of `findServiceData` introspection on WSDL `serviceData` XML elements. This

approach is completely orthogonal to the one taken used in WASP based on asynchronous one-way methods (see Section 5.4).

5.8.6 Registry

Section 5.3.5 has presented an advanced WASP-based Registry service for high-throughput white, yellow, and green page-based Grid service instance discovery.

In order to save development time and also evaluate other implementations, the OGSI-based implementation of ZENTURIO incorporates the VORegistry service provided by the Globus distribution. The uniqueness of the VORegistry is the ability to publish Grid services as service data elements with service lookup support through `findServiceData` introspections. As an additional service data element, the VORegistry publishes a Web Services Inspection Language (WSIL) [10] document containing a list of URLs to all the services registered. The service lookup operations are based on standard XPath [32] queries against the WSIL XML document. Subscriptions on service data element changes provide support push event notifications.

A comparative service lookup throughput analysis for both the WASP Registry and the OGSI VORegistry will be presented in Section 7.1.6.

5.8.7 Security

The user identity in the Grid Security Infrastructure (GSI – see Section 2.4)) is represented by a private and public key pair plus an X.509 certificate. The secure communication across Grid services is realised based on message level WS-Security [9] standard that describes enhancements to SOAP for message integrity through XML digital signatures, message confidentiality through XML encryption, and single message authentication.

The Globus implementation of OGSI includes complete GSI support comprising proxy delegation techniques. The main limitation of the WASP PKI-based security across pure Web services is the support for delegation. The WASP-based Grid services employ the real user private (and public) key for mutual authentication which may be a crucial security flow.

This limitation is critical in two situations in ZENTURIO which illustrate two typical scenarios for which GSI enhancements have been thought:

1. When the Factory (running potentially with administration permissions) creates a new Grid service, it is often natural to give to the newly created instance the identity of the end-user that requested it. This requires that the remote service instance has access to the user private key, which is an unacceptable security risk. Through the GSI delegation mechanism, the Factory provides the service instance with a proxy that impersonates the user for a limited time interval that significantly reduces the security risks;

2. When multiple Grid services are chained in a workflow, they often need to take the client role on behalf on the end-user. Similarly to the Factory case, the proxy delegation achieves this goal with less security risks than propagating the user private key on all Grid sites that host the services from the workflow chain.

The GSI is employed by the WASP implementation too when talking to the Globus services like GASS, GRAM, and GridFTP.

A small test performed for both WASP and OGSI-based implementations shows an increase in latency of about two orders of magnitude with each authenticated call, as opposed to the non-secure version (see Figure 5.17). This high overhead is due to the additional message exchanges between the client and the Grid service needed for performing the mutual authentication.

5.8.8 Summary

Table 5.8.8 presents a comparative summary of the various features which have been comparatively analysed in the WASP and the OGSI-based implementations of ZENTURIO.

Despite the portability limitation, there are some clear advantages of the WASP-based implementation compared to the OGSI-based prototype.

1. WASP generates stubs to remote services dynamically at run-time, which avoids unnecessary compilation steps. OGSI Apache Axis generates stubs statically at compile-time, which restricts the implementation of WSDL complex structures to Java Beans;
2. The WASP-based Factory allows run-time on-the-fly service creation and deployment. This cannot be achieved in the OGSI-based prototype which

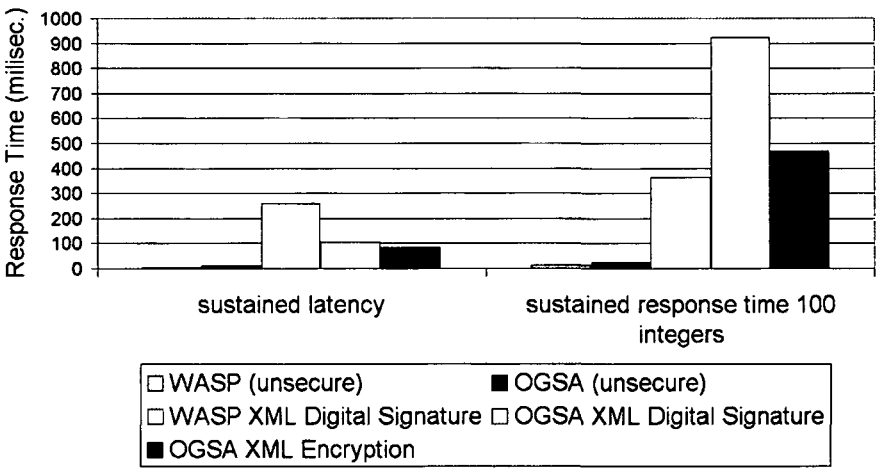


Fig. 5.17. Secure versus unsecure response time comparison.

Functionality	ZENTURIO WASP	ZENTURIO OGSI
WSDL interface	single `portType`	OGSI-compliant
Grid Service	Abstract Grid Service	GridService interface
Registry	yes	yes (VORegistry)
Factory	yes	yes
service creation	dynamic, on-the-fly	static, pre-installed
service lifetime	WASP proprietary	OGSI-compliant
events	WASP-specific Producer/Consumer	OGSI-compliant NotificationSource/Sink
pull events	asynchronous methods	Service Data queries
state introspection	Java Bean access methods	WSDL findServiceData calls
UDDI Repository	yes	no
security	SOAP XML message, no delegation	SOAP XML message, GSI delegation
stubs	run-time, dynamic	static, compile-time
input structures	arbitrarily complex	Java Beans only
service throughput	200 req/sec, 100 int array 400 req/sec, 100 char string	100 req/sec, 100 int array 200 req/sec, 100 char string
registry throughput	Registry service 700 - 300 requests/sec.	VORegistry service 50 - 0.1 requests/second
WSIF support	no	yes

Table 5.5. WASP versus OGSI-based solutions to Grid services features.

restricts the transient service creation to pre-deployed services. This is a severe limitation in a Grid environment where creating services dynamically on unknown remote sites is mandatory;

3. ZENTURIO defines a novel use of the UDDI service repository for storing implementations of transient Grid services;
4. The ZENTURIO WASP-based services provide a better service throughput, which is important in a heavily used multi-client Grid environment (see Section 7.1.7);
5. The ZENTURIO Registry service provides a much better throughput than the OGSI VORegistry (see Section 7.1.6). The reason is the hash-based service organisation of the WASP-based Registry, as opposed to the sequential XML-based service data document of the OGSI-based VORegistry.

6

Optimisation Framework

Chapter 4 has introduced the ZENTURIO experiment management tool for cross-experiment performance and parameter studies of parallel applications. To achieve this goal, ZENTURIO performs an automatic *exhaustive sweep* of the entire parameter space defined using the ZEN directives described in Chapter 3.

With the emergence of Grid computing that aggregates a potentially unbounded number of resources, new classes of applications such as workflows and parameter studies are being defined. The parameter space of such large-scale Grid applications can easily achieve rather huge dimensions for which the exhaustive parameter sweep performed by ZENTURIO is no longer feasible. In general, a complete parameter sweep gives useful detailed insight on the application behaviour, but also produces vast amounts of data that are irrelevant for further studies. Often the user ultimate goal is to find parameter combinations that *optimise* a certain application behaviour, such as a performance metric or an output result. Such optimisation problems are well known as *NP-complete* [74] and require advanced heuristics.

ZENTURIO designs a generic optimisation framework [129] sketched in Figure 6.1 that employs general purpose heuristic algorithms for solving NP-complete performance and parameter optimisation problems for parallel and Grid applications. The input to the optimisation framework consists of a ZEN application and an objective function. The ZEN application defines through ZEN directives a large parameter space impossible to be exhaustively explored. The ZEN application is given as input to a heuristic-based *search engine* that attempts to find a ZEN application instance which maximises the optimisation function. For the realisation of the search engine, general-purpose heuristics like genetic algorithms are considered.

Definition 6.1. *Let $\mathcal{A}$ denote a ZEN application which defines a search space of size $|\mathcal{V}^{\mathcal{A}}|$. The* objective function *to be maximised by the ZENTURIO optimisation framework has the following problem-independent signature:*

$$\mathcal{F} : \mathcal{V}^{\mathcal{A}} \rightarrow \mathbb{R}.$$

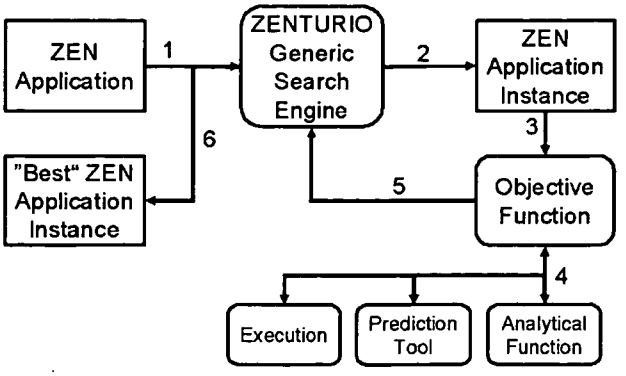


Fig. 6.1. The ZENTURIO optimisation framework design.

The objective function is the only module that depends on the target back-end application which has to be separately supplied for each particular optimisation problem. In the case of performance tuning of parallel applications, the objective function is the performance data defined in Section 3.8 (see Definition 3.26) and evaluated through *experiment execution*. In the case of scheduling problems, the objective function can be implemented by a *performance prediction tool* [54], or approximated through an application-specific *analytical function*. The framework provides a generic objective function interface (see Definition 6.1) that hides the internal problem dependencies, thus keeping the search engine entirely generic.

Within the ZENTURIO architecture described in Chapter 4, the heuristic-based search engine replaces the experiment generation algorithm which was presented in Section 3.10 and encoded as part of the Experiment Generator service (using the random experiment generation method) described in Section 4.2 (see Figure 4.8).

The following three concrete instantiations of the framework will be addressed in this chapter:

1. scheduling single Grid workflow applications in Section 6.2;
2. scheduling large sets of independent tasks for high throughput on the Grid in Section 6.4;
3. optimisation of parallel applications, with special focus on scheduling on heterogeneous Grid resources in Section 6.5.

The next section presents a problem-independent realisation of the optimisation search engine based on genetic algorithms. New general-purpose heuristics like subdivision, simplex, simulated annealing, BFGS, or EPSOC methods will be targeted in future work.

6.1 Genetic Search Engine

Genetic algorithms [77] are a class of randomised optimisation programs which mimic the natural evolution of *individuals* in a *population.* Genetic algorithms use a vocabulary borrowed from natural genetics. Often individuals are called *chromosomes.* Chromosomes are made of units called *genes,* arranged in linear succession. Genes are located at certain places in the chromosome called *loci.* The value of a gene which determines one character of an individual (such as hair colour) is called *allele.* The genetic algorithms are iterative algorithms that start from an initial population and use natural evolution operators on the population individuals. The *selection* operator selects some better fit individuals from the population according to a *fitness function.* The selected individuals then qualify for *reproduction*, *crossover*, and *mutation* with certain probabilities. As a result, a new population of more superior individuals is obtained. The iterative process continues on the newly formed population until a convergence criterion is fulfilled.

ZENTURIO employs a classical generational genetic algorithm sketched in Figure 6.2. This section presents a generic encoding of the genetic search engine that is independent of the objective function and therefore can be applied to multiple optimisation problems. The optimisation function that implements a generic API interface represents an application-dependent performance metric or output result which has to be independently supplied.

algorithm *genetic optimiser*;
input: (1) ZEN application $\mathcal{A}$;
(2) fitness function $\mathcal{F}$;
(3) population size p;
(4) crossover probability p_c;
(5) mutation probability p_m;
(6) maximum generation *max_gen*;
(7) steady state percentage;
(8) fitness scaling factor C_{mult};
(9) elitist model;
output: "best" ZEN application instance $\mathcal{AI}$;

1. *Initialise* the population with individuals;
2. *generation* = 1;
   repeat
3. *Select* and *reproduce* ZEN application instances;
4. *Crossover* ZEN application instances with probability p_c;
5. *Mutate* each ZEN variable with probability p_m;
6. *generation* = *generation* + 1;
   until *convergence criterion* **or** *steady state* **or** *generation* $\geq$ *max_gen*;
7. **return** the ZEN application instance with maximum fitness value.

Fig. 6.2. The generational genetic algorithm.

Definition 6.2. *Let* $\mathcal{A}(z_1, \ldots, z_n)$ *denote a ZEN application, where* z_i *are ZEN variables,* $\forall\ i \in [1..n]$. *Let* $\mathcal{V}^{z_i}$ *denote the value set of a ZEN variable* z_i *(i.e., the set of possible parameter instantiation values). A* gene *is a ZEN variable* z_i. *An* allele *is a gene instantiation, i.e., an element* $e_i \in \mathcal{V}^{z_i}$. *The totally ordered set* $\{z_1, \ldots, z_n\}$ *of all ZEN variables of* $\mathcal{A}$ *is a* chromosome. *The* locus i *of a gene* z_i *is given by its index within the totally ordered set chromosome. An* individual *is a ZEN application instance* $\mathcal{AI}(e_1, \ldots, e_n)$, *where* $e_i \in \mathcal{V}^{z_i}$, $\forall\ i \in [1..n]$. *The* objective function, *called in genetic terms* fitness function, *has been defined in Definition 6.1.*

6.1.1 Initial Population

The initial population of fixed size p is built by generating a random set of ZEN application instances (i.e., by assigning random values to ZEN variables):

$$\mathcal{P} = \{\mathcal{AI}_i(e_1, \ldots, e_n) \mid e_j \in \mathcal{V}^{z_j},\ \forall\ j \in [1..n],\ \forall\ i \in [1..p]\}.$$

An appropriate population size p has to be experimentally determined for each particular problem. Additionally, an interface for manually inserting ZEN application instances in the initial population, which can significantly improve the performance of the genetic algorithm, is provided.

6.1.2 Selection

The selection operator creates a new population by choosing the best ZEN application instances for reproduction. Let $\mathcal{P}$ denote a population of cardinality p and $\overline{\mathcal{F}}$ its average fitness. The *reminder stochastic sampling with replacement* [77] selection model that creates a new population:

$$\mathcal{P}' = \mathcal{P}_1 \cup \mathcal{P}_2$$

in two steps, as follows:

1. $\mathcal{P}_1 = \bigcup_{i=1}^{p} \bigcup_{j=1}^{\left\lceil \frac{\mathcal{F}(\mathcal{AI}_i)}{\overline{\mathcal{F}}} \right\rceil} clone_j(\mathcal{AI}_i)$. This step is called *expected value model* because it selects each application instance proportional with its fitness value and eliminates stochastic sampling errors;
2. $\mathcal{P}_2 = \bigcup_{i=1}^{s} clone_j(\mathcal{AI}_j)$, where $s = |\mathcal{P}| - |\mathcal{P}_1|$, $r_i \in [0, 1]$ is a random number such that:

$$\frac{\sum_{k=1}^{j-1} \left\{ \frac{\mathcal{F}(\mathcal{AI}_k)}{\overline{\mathcal{F}}} \right\}}{\sum_{k=1}^{p} \left\{ \frac{\mathcal{F}(\mathcal{AI}_k)}{\overline{\mathcal{F}}} \right\}} < r_i \leq \frac{\sum_{k=1}^{j} \left\{ \frac{\mathcal{F}(\mathcal{AI}_k)}{\overline{\mathcal{F}}} \right\}}{\sum_{k=1}^{p} \left\{ \frac{\mathcal{F}(\mathcal{AI}_k)}{\overline{\mathcal{F}}} \right\}},$$

and $|\mathcal{P}|$ denotes the cardinality of the set $\mathcal{P}$. Informally, the population places that remained empty in the first step are filled by simulating a roulette wheel with slots proportional with the fractional part of each individual fitness normalised against the average population fitness.

6.1.3 Crossover

The crossover operator is used in genetic algorithms for performing quick searches for local maxima. The algorithm employs a *single point crossover operator*, defined by the random function:

$$\oplus_r : \mathcal{V}^{\mathcal{A}} \times \mathcal{V}^{\mathcal{A}} \rightarrow \mathcal{V}^{\mathcal{A}} \times \mathcal{V}^{\mathcal{A}},$$
$$\mathcal{AI}_1(e_1, \ldots, e_n) \oplus \mathcal{AI}_2(e'_1, \ldots, e'_n) = (\mathcal{AI}'_1, \mathcal{AI}'_2),$$

where:

$$\mathcal{AI}'_1 = \mathcal{AI}'_1(e_1, \ldots, e_r, e'_{r+1}, \ldots, e'_n),$$
$$\mathcal{AI}'_2 = \mathcal{AI}'_2(e'_1, \ldots, e'_r, e_{r+1}, \ldots, e_n),$$

and $r \in [1, n-1]$ is a random number (see Figure 6.3(a)).

Let $\mathcal{P} = \{\mathcal{AI}_1, \ldots, \mathcal{AI}_n\}$ denote a population of ZEN application instances. Let p_c be the probability of crossover that has to be experimentally determined for each individual problem. The subset of ZEN application instances which undergo crossover is given by:

$$\mathcal{P}_c = \bigcup_{i=1}^{n} \mathcal{AI}'_i,$$

where:

$$\mathcal{AI}'_i = \begin{cases} \mathcal{AI}_i, & r_i < p_c \\ \Phi, & r_i \geq p_c, \end{cases}$$

and $r_i \in [0, 1]$ is a random number, $\forall\ i \in [1..n]$. The crossover pairs are randomly selected from $\mathcal{P}_c$.

6.1.4 Mutation

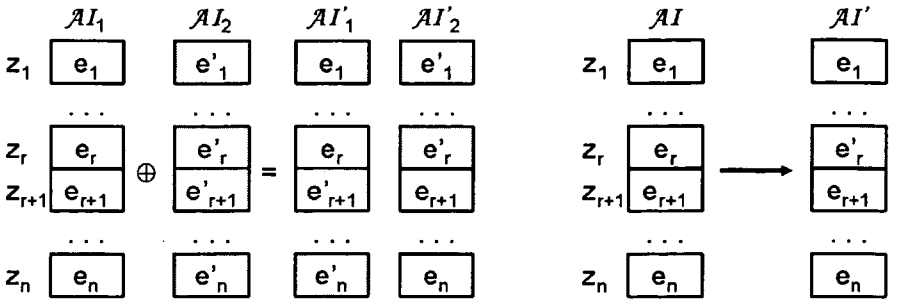
The mutation operator enables the algorithm to jump to another search space region which avoids local stagnation stages of the population. The *mutation* operator applies gene-wise on ZEN application instances, according to the function:

$$\ominus : \mathcal{V}^{\mathcal{A}} \rightarrow \mathcal{V}^{\mathcal{A}},\ \ominus(\mathcal{AI}(e_1, \ldots, e_n)) = \mathcal{AI}'(e'_1, \ldots, e'_n),$$

where:

$$e'_i = \begin{cases} e''_i, & r_i < p_m; \\ e_i, & r_i \geq p_m, \end{cases}$$

p_m is the (experimentally tuned) probability of mutation for a gene, $r_i \in [0, 1]$ is a random number, and $e''_i \in \mathcal{V}^{z_i}$ is a randomly selected allele, $\forall\ i \in [1..n]$. A sample chromosome which undergoes a single gene mutation is illustrated in Figure 6.3(b).



(a) Crossover. (b) Mutation.

Fig. 6.3. The genetic operators.

6.1.5 Elitist Model

Repeated crossover and mutation may lead to the elimination of the best ZEN application instance, which could have negative impacts on the final solution.

Let $\mathcal{P}_G$ be a population at some generation G, $\mathcal{AI}_B^G \in \mathcal{P}_G$ the currently best ZEN application instance $(i.e., \mathcal{F}(\mathcal{AI}_B^G) \geq \mathcal{F}(\mathcal{AI}),\ \forall\ \mathcal{AI} \in \mathcal{P}_G)$, and $\mathcal{P}_{G+1}$ the next generation. The *elitist model* enforces to preserve the best ZEN application instance across generations:

$$\mathcal{P}'_{G+1} = \begin{cases} \mathcal{P}_{G+1}, & \mathcal{AI}_B^{G+1} \geq \mathcal{AI}_B^G; \\ \mathcal{P}_{G+1} - \mathcal{AI} \cup \mathcal{AI}_B^G, & \mathcal{AI}_B^{G+1} < \mathcal{AI}_B^G, \end{cases}$$

where $\mathcal{AI} \in \mathcal{P}_{G+1}$ is a randomly eliminated individual. The elitist model may lead to pre-mature convergence of the algorithm if not carefully applied.

6.1.6 Fitness Scaling

There are two problems with the selection method described in Section 6.1.2:

1. At the start of the algorithm it is common to have several super-individuals (but globally average) that would dominate the later generations and lead to fast pre-mature convergence of the algorithm;
2. Late in the run, the population average fitness often gets close to the best fitness. In this case, average and best members get equally represented in the future generations and the survival of the fittest chromosome necessary for improvement becomes a random walk among the mediocre.

Let $\overline{\mathcal{F}}$ denote the average population fitness. *Linear fitness scaling* defines a new scaled fitness function for one ZEN application instance:

$$\mathcal{F}' = a \cdot \mathcal{F} + b,$$

where a and b are determined by solving the following system of equations:

$$\begin{cases} a \cdot \overline{\mathcal{F}} + b = \overline{\mathcal{F}} \\ a \cdot \mathcal{F}_{max} + b = C_{mult} \cdot \overline{\mathcal{F}}. \end{cases}$$

The two equations insure two crucial aspects for proper genetic algorithm convergence:

1. average scaled fitness $\overline{\mathcal{F}'}$ is equal with the average raw fitness $\mathcal{F}$ because each average ZEN application instance is expected to contribute with one offspring to the next generation;
2. the best ZEN application instance $\mathcal{F}_{max}$ is expected to contribute with C_{mult} offsprings to the next generation. This reduces the gap between super and average individuals in initial generations (which avoids premature convergence) and increases this gap in late generations (which ensures strong competition necessary for continuous healthy survival and improvement).

6.1.7 Convergence Criterion

For flexibility reasons, the algorithm defines three convergence criteria which can be freely combined:

1. when a user-defined convergence criterion (defined by objective function interface) is fulfilled (e.g., fitness value increases above threshold);
2. after a predefined maximum number generations;
3. when a steady state stagnation is achieved after which no further improvements are being made. The steady state is checked by examining the fitness function of the best individual within a sliding window of a predefined number of generations (i.e., percentage from the maximum generation number).

6.2 Static Workflow Scheduling

Workflow modelling is a well established area in computer science that has been strongly influenced by business process modelling work [173]. Recently, the Grid community has become increasingly interested in this topic, as workflow applications define an important class of Grid applications for which several development environments are currently being built [19, 52, 92, 94, 101, 113, 154, 167]. The workflow scheduling problem addressed in this section is based on the workflow model presented in Section 2.8.2.

On computational Grids there are two distinct aspects related to the general single workflow scheduling problem:

1. *static scheduling* or initial (launch-time) scheduling targets the optimal mapping of an entire workflow application onto a fixed set of resources. This problem will be addressed in this section as an instantiation of the ZENTURIO optimisation framework using genetic algorithms;
2. *dynamic scheduling* is a steering problem that adapts the workflow static schedule to the dynamic availability of Grid resources, which will be addressed in Section 6.3. The workflow mapping onto the Grid resources may change during the workflow execution.

The static scheduling of a workflow of n tasks onto m computational Grid resources is a well known NP-complete optimisation problem of $\mathcal{O}(m^n)$ complexity [165]. Since in practice Directed Graph (DG)-based workflow cycles have either large iteration counts or depend on run-time application data, it is problematic to consider them for static scheduling. Rather, the scope of static scheduling is constrained to Directed Acyclic Graph (DAG)-based workflows. The constraint will be relaxed by the dynamic scheduling approach in Section 6.3.

The following definition specifies the instantiation of the static workflow scheduling problem within the ZENTURIO optimisation framework.

Definition 6.3. *A* ZEN variable (gene) z *is an application parameter that represents an abstract Grid machine. A* ZEN application $\mathcal{A}(z_1, \ldots, z_n) = (Nodes, Edges)$ *implements a workflow as defined by Definition 2.7, where:*

1. $\forall\ JS(z) \in Nodes \implies z \in \{z_1, \ldots, z_n\}$;
2. $\forall\ FT(z, z') \in Nodes \implies \{z, z'\} \subset \{z_1, \ldots, z_n\}$.

The value set $\mathcal{V}^{z_i}$ *of a ZEN variable* z_i *represents the entire set of concrete Grid machines. A* workflow schedule *or an* individual *is a mapping:*

$$\mathcal{S}_{\mathcal{A}} = \mathcal{S}(\mathcal{A}(z_1, \ldots, z_n)) = \mathcal{AI}(e_1, \ldots, e_n),\ \forall\ e_i \in \mathcal{V}^{z_i},\ \forall\ i \in [1..n].$$

Within $\mathcal{AI}$, *a* job submission schedule *is a mapping:*

$$\mathcal{S}_{JS(z_j)} = e_j$$

and a file transfer schedule *is a mapping:*

$$\mathcal{S}_{FT(z_k, z_l)} = (e_k, e_l).$$

Finding the workflow schedule that maximises the objective function is the static workflow scheduling problem.

Example 6.4 sketches an implementation of the workflow depicted in Figure 6.4 based the Java CoG package [7]. The workflow is defined as a ZEN application denoted as $\mathcal{A}(z_1, z_2, z_3, z_4, z_5)$, where z_1, z_2, z_3, z_4, and z_5 are the abstract machines where the workflow tasks are to be scheduled. The ZEN directives that annotate the workflow define the set of possible concrete machines (with cardinality 100) that instantiate each abstract machine within a workflow schedule.

Example 6.4 (Java DAG-based workflow).

```
//ZEN$ SUBSTITUTE z1 = { e{1:100} }
//ZEN$ SUBSTITUTE z2 = { e{1:100} }
//ZEN$ SUBSTITUTE z3 = { e{1:100} }
//ZEN$ SUBSTITUTE z4 = { e{1:100} }
//ZEN$ SUBSTITUTE z5 = { e{1:100} }

Task js1 = createJS(''z1'');
Task js2 = createJS(''z2'');
Task js3 = createJS(''z3'');
Task ft4 = createFT(''z1'', ''z4'');
Task js5 = createJS(''z4'');
Task ft6 = createFT(''z4'', ''z1'');
Task js7 = createJS(''z5'');
Task js8 = createJS(''z1'');

TaskGraph taskGraph = new TaskGraphImpl();
taskGraph.add(js1);
taskGraph.add(js2);
taskGraph.add(js3);
taskGraph.add(ft4);
taskGraph.add(js5);
taskGraph.add(ft6);
taskGraph.add(js7);
taskGraph.add(js8);

Dependency dependency = new DependencyImpl();
dependency.add(js1, js2);
dependency.add(js1, js3);
dependency.add(js1, ft4);
dependency.add(js2, js5);
dependency.add(js3, js5);
dependency.add(ft4, js5);
dependency.add(js5, ft6);
dependency.add(js5, js7);
dependency.add(ft6, js8);
dependency.add(js7, js8);
taskGraph.setDependency(dependency);
```

6.2.1 Genetic Static Scheduler

The static scheduler employs the general-purpose heuristics proposed by the ZENTURIO optimisation framework. The following definition specifies the instantiation of the genetic algorithm described in Section 6.1 for workflow applications.

Definition 6.5. *Let $\mathcal{A}(z_1, \ldots, z_n)$ denote a ZEN application that represents a workflow application as defined in Definition 6.3. A* gene *z is a ZEN variable that represents an abstract Grid machine. An* allele *$e_i \in \mathcal{V}^{z_i}$ is a concrete Grid machine. The totally ordered set $(\{z_1, \ldots, z_n\}, \prec)$ builds a* chromosome, *where the total order $\prec$ of genes in a chromosome (i.e.,* loci*) is fixed and respects the (partial) node topological order:*

$$N_i \prec N_j \Rightarrow N_i \notin succ^p(N_j).$$

Figure 6.4 illustrates two sample crossover and mutation operations for the workflow application encoded in Example 6.4.

6.2.2 Schedule Dependencies

Definition 6.6. *If the same ZEN variable or abstract machine appears in the definition of two distinct tasks (see Definition 2.7), it defines a* static schedule dependency *(e.g., $JS_1(z_1), FT_4(z_1, z_4), FT_6(z_4, z_1), JS_8(z_1)$ in Figure 6.4(b)).*

A typical example is the abstract machine z_4 in Figure 6.4(b), where the task $JS_5(z_4)$ stages in its input file from the machine z_1 through the task $FT_4(z_1, z_4)$ and stages out its output file to the machine z_1 through the task $FT_6(z_4, z_1)$. The mutation of a gene involved in a static schedule dependency has the effect shown in Figure 6.4(b) for the abstract machine z_4. Static schedule dependencies can also be set between the *JS* tasks. In Figure 6.4(b) for instance, the tasks $JS_1(z_1)$ and $JS_8(z_1)$ define a static schedule dependence that restricts their schedule to the same concrete machine, i.e., $\mathcal{S}_{JS_1(z_1)} = \mathcal{S}_{JS_8(z_1)}$.

Definition 6.7. *Let $JS_1, \ldots, JS_m$ be a set of independent job submission tasks, such that $\mathcal{S}_{JS_1} = \ldots = \mathcal{S}_{JS_m}$. A* valid workflow schedule *is obtained by augmenting the original workflow application with* run-time schedule dependencies *that prohibit two independent job submission tasks run on the same machine concurrently:*

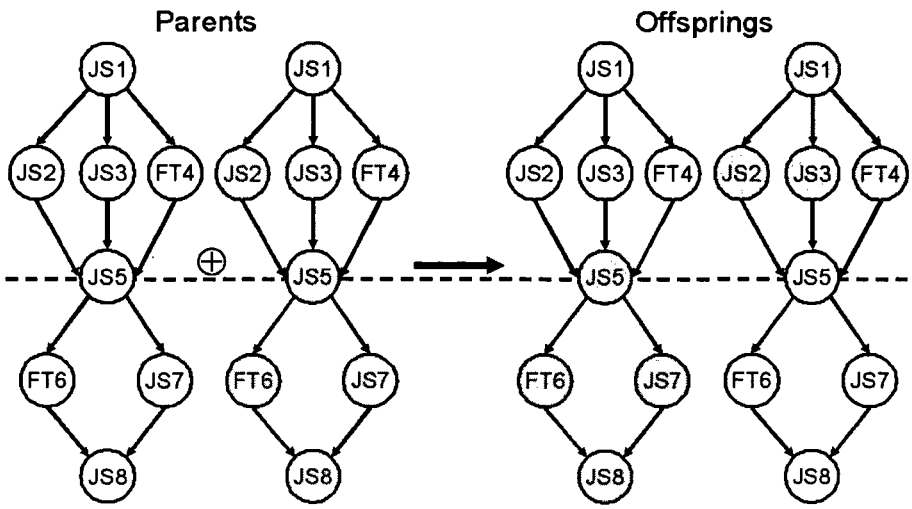
$$Edges' = Edges \cup \{(JS_i, JS_{i+1}) \mid \mathcal{S}_{JS_i} = \mathcal{S}_{JS_{i+1}}, \forall\, i \in [1..m-1]\}.$$

Figure 6.4(b) illustrates such a run-time schedule dependency between the tasks JS_2 and JS_3, assuming that $\mathcal{S}_{JS_2} = \mathcal{S}_{JS_3}$, which augments the set of workflow edges as follows:

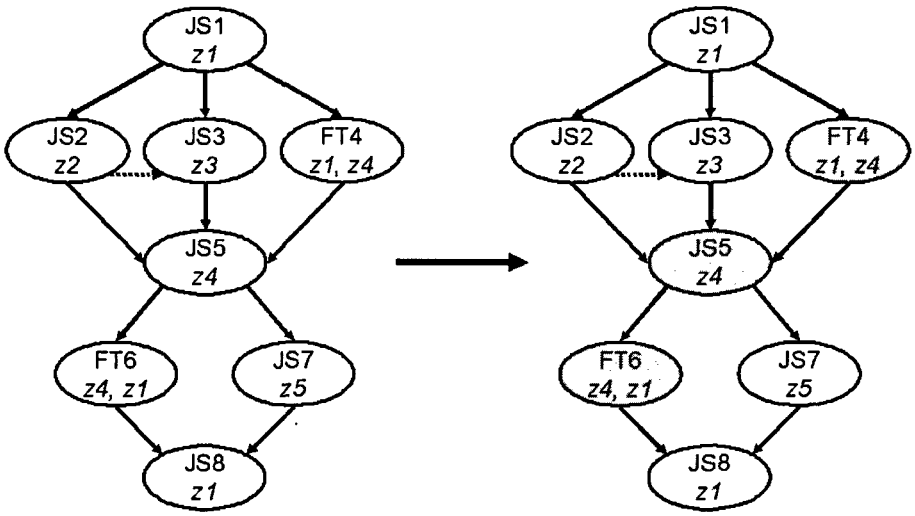
$$Edges' = Edges \cup \{(JS_2, JS_3)\}.$$

Lemma 6.8. *Given a workflow schedule, to determine the optimum valid schedule is an NP-complete problem.*

Proof. If m independent tasks are scheduled on the same abstract machine, there are $m!$ possible run-time schedule dependencies. Scheduling p sets of m_i independent tasks each such that $\mathcal{S}(N_{j1}) = \ldots = \mathcal{S}(N_{jm_i})$, $\forall\, j \in [1..p]$ is a classical NP-complete scheduling problem [90].



(a) Crossover.



(b) Mutation.

Fig. 6.4. The workflow genetic operators.

Since in practice the number of computational Grid resources is normally much larger than the number of workflow tasks, the probability of having large sets of run-time schedule dependencies is low. In addition, since the static scheduler is based on randomised heuristics, a random valid schedule can be selected without altering the quality of the final solution. In the context of the genetic algorithm, if a clone of an existing individual (i.e., identical schedule) is produced as a result of crossover or mutation (not as an outcome of selection), a different random valid schedule is selected, thus insuring the population diversity necessary for improvement.

6.2.3 Objective Function

The objective function for the static scheduling problem is represented by a performance metric to be optimised. The computation of workflow performance metrics for scheduling purposes relies on existing *prediction* models for each individual task, which is a difficult research topic [54] that goes beyond the scheduling work targeted by this thesis.

Definition 6.9. *Let N be an arbitrary task with the schedule S_N. The* predicted execution time *of N onto S_N is approximated as:*

$$T_N^{S_N} = \frac{W_N}{v_{S_N}},$$

where W_N stands for the work *of task N and v_{S_N} for the* speed *of S_N with the following semantics:*

1. *W_{JS} represents the number of floating point operations of task JS and $v_{S_{JS}}$ represents the performance rate of the machine S_{JS} (e.g., as returned by the LINPACK [47] benchmark);*
2. *W_{FT} approximates the file size and $v_{S_{FT}}$ the bandwidth of a single TCP stream between e_1 and e_2, where $S_{FT} = (e_1, e_2)$.*

Since the Grid workflows are used to model course grain composition of large applications distributed over wide-area networks, the latencies between dependent tasks can be safely ignored. Section 7.3.1 will present a concrete instantiation of this rather trivial prediction model for a real world application.

Definition 6.10. *Let $(Nodes, Edges)$ denote a workflow application. A workflow schedule is evaluated by constructing the* Gantt chart *that simulates the workflow execution. The end timestamp of each workflow task $N \in Nodes$ is recursively defined by the following function:*

$$end : Nodes \to \mathbb{N},\ end(N) = \begin{cases} T_N^{S_N}, & pred(N) = \phi; \\ \max\limits_{(N',N) \in Edges} \{end(N')\} + T_N^{S_N}, & pred(N) \neq \phi, \end{cases}$$

where $\mathbb{N}$ denotes the set of natural numbers.

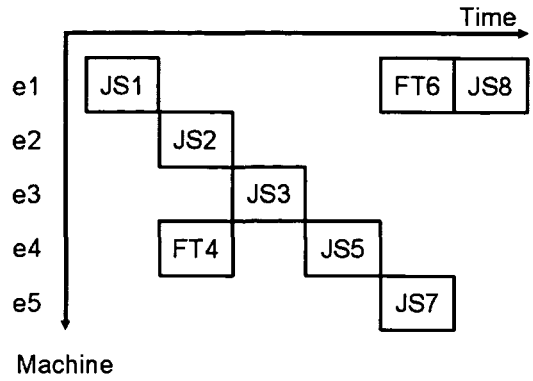


Fig. 6.5. Sample Gantt chart for the workflow depicted in Figure 6.4(b), assuming that $e_2 = e_3$ (i.e., $\mathcal{S}_{JS_2} = \mathcal{S}_{JS_3}$).

Figure 6.5 illustrates a sample Gantt chart for the workflow depicted in Figure 6.4, assuming the run-time schedule dependency (JS_2, JS_3) (i.e., $\mathcal{S}_{JS_2} = \mathcal{S}_{JS_3}$), where:

$$
\begin{aligned}
end(JS_1) &= T^{e_1}_{JS_1};\\
end(JS_2) &= end(JS_1) + T^{e_2}_{JS_2};\\
end(JS_3) &= end(JS_2) + T^{e_3}_{JS_3};\\
end(FT_4) &= end(JS_1) + T^{(e_1,e_4)}_{JS_3};\\
end(JS_5) &= max\{end(JS_2), end(JS_3), end(FT_4)\} + T^{e_4}_{JS_5};\\
end(FT_6) &= end(JS_5) + T^{(e_4,e_1)}_{FT_6};\\
end(JS_7) &= end(JS_5) + T^{e_5}_{JS_7};\\
end(JS_8) &= max\{end(FT_6), end(JS_7)\} + T^{e_1}_{JS_8}.
\end{aligned}
$$

Definition 6.11. *Let* $\rho = \{N_1, \ldots, N_p\}$ *denote a* workflow execution path, *i.e.,* $pred(N_1) = \phi \ \wedge \ succ(N_p) = \phi \ \wedge \ (N_i, N_{i+1}) \in Edges, \ \forall \ i \in [1..p-1]$. *If* N_p *is the task with the maximum end time and* ρ *is the shortest path to* N_p, *then* ρ *is called the* critical schedule path:

1. $end(N_p) = \underset{N \in Nodes \ \wedge \ succ(N)=\phi}{max} \{end(N)\}$;
2. $\sum_{\forall N \in \rho} T^{\mathcal{S}_N}_N \leq \sum_{\forall N' \in \rho'} T^{\mathcal{S}_{N'}}_{N'}, \ \forall \ \rho' = \{N'_1, \ldots, N'_q\}$ *a workflow execution path (i.e.,*$pred(N'_1) = \phi \ \wedge \ succ(N'_q) = \phi \ \wedge \ (N'_i, N'_{i+1}) \in Edges, \ \forall \ i \in [1..q-1]$*), such that* $end(N_p) = end(N'_q)$.

Let $\mathcal{A} = \left(Nodes = Nodes^{JS} \cup Nodes^{FT}, Edges\right)$ denote a workflow application, $\mathcal{AI}(e_1, \ldots, e_n)$ a workflow schedule, $z = \bigcup_{i=1}^{n} e_i$ the set of underlying concrete machines of $\mathcal{AI}$, and $|z|$ the cardinality of the set z. In the following, a

range of sample objective functions representing useful workflow performance metrics are defined. Since the framework has been designed to solve maximum problems, some of the workflow metrics that require minimisation had to be subtracted from a large enough constant C.

- *Makespan* or execution time:

$$\mathcal{F}(\mathcal{AI}) = C - T_{\mathcal{AI}},$$
$$T_{\mathcal{AI}} = end(N_p),$$

  where $\{N_1, \ldots, N_p\}$ is the critical schedule path;
- *Speedup:*

$$\mathcal{F}(\mathcal{AI}) = S_{\mathcal{AI}},$$
$$S_{\mathcal{AI}} = \frac{T^{seq}_{\mathcal{AI}}}{T_{\mathcal{AI}(e_1,\ldots,e_n)}},$$
$$T^{seq}_{\mathcal{AI}} = \min_{\forall i \in [1..n]} \left\{ T_{\mathcal{AI}(e_i,\ldots,e_i)} \right\},$$
$$T_{\mathcal{AI}(e_i,\ldots,e_i)} = \sum_{\forall JS \in Nodes^{JS}} T^{e_i}_{JS},$$

  where $\mathcal{AI}(e_i, \ldots, e_i)$ is the sequential workflow schedule on machine e_i;
- *Efficiency:*

$$\mathcal{F}(\mathcal{AI}) = E_{\mathcal{AI}},$$
$$E_{\mathcal{AI}} = \frac{S_{\mathcal{AI}}}{|z|};$$

- *Communication* due to file transfer tasks on the critical path:

$$\mathcal{F}(\mathcal{AI}) = C - C_{\mathcal{AI}},$$
$$C_{\mathcal{AI}} = \sum_{\forall N \in \rho \cap Nodes^{FT}} T^{S_N}_N,$$

  where ρ is the critical schedule path;
- *Synchronisation* due to task dependencies on the critical path:

$$\mathcal{F}(\mathcal{AI}) = C - SY_{\mathcal{AI}},$$
$$SY_{\mathcal{AI}} = T_{\mathcal{AI}} - \sum_{\forall N \in \rho} T^{S_N}_N,$$

  where ρ is the critical schedule path;
- *Load Balance* due to uneven work distribution:

$$\mathcal{F}(\mathcal{AI}) = LB_{\mathcal{AI}},$$
$$LB_{\mathcal{AI}} = \frac{\underset{\forall e \in z}{avg} \left\{ \sum_{\{\forall JS \in Nodes^{JS} | S_{JS} = e\}} T^e_{JS} \right\}}{\underset{\forall e \in z}{max} \left\{ \sum_{\{\forall JS \in Nodes^{JS} | S_{JS} = e\}} T^e_{JS} \right\}}.$$

$LB_{\mathcal{AI}} = 1$ indicates the perfect load balance and $LB_{\mathcal{AI}} = 0$ the worst case load balance;

- *Total Overhead* defined by the Amdahl's law [6]:

$$\begin{aligned}\mathcal{F}(\mathcal{AI}) &= \mathcal{C} - O_{\mathcal{AI}},\\ O_{\mathcal{AI}} &= T_{\mathcal{AI}} - \frac{T^{seq}_{\mathcal{AI}}}{|z|}\\ &= \sum_{N \in \rho \cap Nodes^{JS}} T^{S_N}_N + C_{\mathcal{AI}} + SY_{\mathcal{AI}} - \frac{T^{seq}_{\mathcal{AI}}}{|z|};\end{aligned}$$

- *Loss of Parallelism* due to heterogeneity and task dependencies on the critical path:

$$\begin{aligned}\mathcal{F}(\mathcal{AI}) &= \mathcal{C} - LP_{\mathcal{AI}},\\ LP_{\mathcal{AI}} &= O_{\mathcal{AI}} - C_{\mathcal{AI}} - SY_{\mathcal{AI}}\\ &= \sum_{N \in \rho \cap Nodes^{JS}} T^{S_N}_N - \frac{T^{seq}_{\mathcal{AI}}}{|z|};\end{aligned}$$

- *Efficiency + Execution time.* Maximising efficiency combined with minimising execution time is a good metric for high *throughput scheduling*, in the context of multiple workflows (super- or meta-scheduling).

These metrics can be instantiated for the workflow defined in Example 6.4 and depicted in Figure 6.4(b) as follows:

- $T_{\mathcal{AI}} = end(JS_8)$;
- $S_{\mathcal{AI}} = \frac{\min\limits_{\forall i \in \{1,2,3,4,5\}} \{T_{\mathcal{AI}(e_i,\ldots,e_i)}\}}{end(JS_8)}$, where $T_{\mathcal{AI}(e_i,\ldots,e_i)} = T^{e_i}_{JS_1} + T^{e_i}_{JS_2} + T^{e_i}_{JS_3} + T^{e_i}_{JS_5} + T^{e_i}_{JS_7} + T^{e_i}_{JS_8}$;
- $E_{\mathcal{AI}} = \frac{S_{\mathcal{AI}}}{5}$;
- $C_{\mathcal{AI}} = T^{(e_1,e_4)}_{FT_4} + T^{(e_4,e_1)}_{FT_6}$;
- $SY_{\mathcal{AI}} = T_{\mathcal{AI}} - T^{e_1}_{JS_1} - T^{e_2}_{JS_2} - T^{e_4}_{JS_5} - T^{(e_4,e_1)}_{FT_6} - T^{e_1}_{JS_8}$, where the workflow path $(JS_1, JS_2, JS_5, FT_6, JS_8)$ is the critical schedule path, which assumes that the following conditions hold:

$$T^{e_1}_{JS_1} + T^{e_2}_{JS_2} + T^{e_4}_{JS_5} + T^{(e_4,e_1)}_{FT_6} + T^{e_1}_{JS_8} \leq T^{e_1}_{JS_1} + T^{e_3}_{JS_3} + T^{e_4}_{JS_5} + T^{(e_4,e_1)}_{FT_6} + T^{e_1}_{JS_8};$$
$$T^{e_1}_{JS_1} + T^{e_2}_{JS_2} + T^{e_4}_{JS_5} + T^{(e_4,e_1)}_{FT_6} + T^{e_1}_{JS_8} \leq T^{e_1}_{JS_1} + T^{(e_1,e_4)}_{FT_4} + T^{e_4}_{JS_5} + T^{(e_4,e_1)}_{FT_6} + T^{e_1}_{JS_8};$$
$$T^{e_1}_{JS_1} + T^{e_2}_{JS_2} + T^{e_4}_{JS_5} + T^{(e_4,e_1)}_{FT_6} + T^{e_1}_{JS_8} \leq T^{e_1}_{JS_1} + T^{e_2}_{JS_2} + T^{e_4}_{JS_5} + T^{e_5}_{JS_7} + T^{e_1}_{JS_8};$$
$$T^{e_1}_{JS_1} + T^{e_2}_{JS_2} + T^{e_4}_{JS_5} + T^{(e_4,e_1)}_{FT_6} + T^{e_1}_{JS_8} \leq T^{e_1}_{JS_1} + T^{e_3}_{JS_3} + T^{e_4}_{JS_5} + T^{e_5}_{JS_7} + T^{e_1}_{JS_8};$$
$$T^{e_1}_{JS_1} + T^{e_2}_{JS_2} + T^{e_4}_{JS_5} + T^{(e_4,e_1)}_{FT_6} + T^{e_1}_{JS_8} \leq T^{e_1}_{JS_1} + T^{(e_1,e_4)}_{FT_4} + T^{e_4}_{JS_5} + T^{e_5}_{JS_7} + T^{e_1}_{JS_8};$$

- $LB_{\mathcal{AI}} = \frac{avg\left\{T^{e_1}_{JS_1} + T^{e_1}_{JS_8}, T^{e_3}_{JS_3}, T^{e_4}_{JS_5}, T^{e_5}_{JS_7}\right\}}{max\left\{T^{e_1}_{JS_1} + T^{e_1}_{JS_8}, T^{e_3}_{JS_3}, T^{e_4}_{JS_5}, T^{e_5}_{JS_7}\right\}}$.

The two remaining metrics (i.e., total overhead and loss of parallelism) derive from these five.

6.3 Dynamic Workflow Scheduling

The static workflow scheduling approach described in Section 6.2 suffers of two limitations:

1. loops are not comprised by the DAG-based workflow model;
2. the Grid is not considered as a dynamic environment where the resources can change run-time load and availability.

This section presents a hybrid dynamic scheduling algorithm that is based on the repeated invocation of the static scheduler, as informally outlined by the following execution steps.

1. The algorithm receives as input a *DG-based workflow* that may include loops;
2. A *cycle elimination* operation transforms the DG-based workflow into a DAG, by eliminating the edges oriented against the node topological order;
3. The generated DAG is given as input to the *static scheduling algorithm* for optimised mapping onto the set of available Grid resources;
4. The mapping of the DAG completely defines a (static) workflow schedule which is submitted for *execution*;
5. The workflow execution is *monitored* at well-defined scheduling events generated at a frequency that depends on the load variation of the available Grid resources;
6. At each scheduling event, the execution of every running task is evaluated according to the performance contract developed by the static scheduler (see Definition 6.13). If the evaluation is negative, the task is selected for *migration* and requires *rescheduling*;
7. Based on the workflow execution status, a *new DAG* is generated according to the rules formally defined in Section 6.3.2;
8. The monitoring and the static scheduling algorithm are *repeated* at the scheduling event frequency until the workflow execution has completed.

Definition 6.12. *A task $N \in$ Nodes of the running workflow (Nodes, Edges) can be at a certain time instance t in one of the* states *queued, running, completed, or failed, denoted as state(N, t).*

The reminder of this section formally defines the task migration conditions (see Section 6.3.1 and the static DAG generation process (see Section 6.3.2). The dynamic scheduling algorithm informally outlined so far is depicted in Figure 6.6 in self-explanatory pseudo-code.

6.3.1 Task Migration

Let N be a submitted task, W_N its underlying work assigned (i.e., floating point operations for *JS* tasks, file size for *FT* tasks), $T_N^{S_N}$ its estimated execution time, and

algorithm *dynamic scheduler*;
input: *workflow:* $\mathcal{A} = (Nodes, Edges)$;
  cycle elimination: $\mathcal{A}_0 = (Nodes, Edges - Edges_{Queued})$
    ($Edges_{Queued}$ is defined in Section 6.3.2);
  static schedule: $\mathcal{AI}$ = *genetic optimiser*($\mathcal{A}_0$);
  submit workflow: execute($\mathcal{A}, \mathcal{AI}$);
  repeat
    t = **sleep** until next scheduling event;
    select tasks for migration:
      $Nodes_{Migr} = \{N \in Nodes \mid state(N,t) = failed \vee$
        $state(N,t) = running \wedge PC(N, \mathcal{S}_N, t) > f_N\}$
    $\mathcal{A}_t$ = *generate static* $DAG(\mathcal{A}, \mathcal{AI}, t, Nodes_{Migr})$;
    cancel$(N), \forall N \in Nodes_{Migr}$;
    static reschedule: $\mathcal{AI}$ = *genetic optimiser*($\mathcal{A}_t$);
  until $state(N,t) = completed, \forall N \in Nodes \wedge succ(N) = \phi$;

Fig. 6.6. The dynamic scheduling algorithm.

$$start(N) = end(N) - T_N^{\mathcal{S}_N}$$

its start timestamp, where $end(N)$ has been defined in Section 6.2.3 (see Definition 6.10).

Definition 6.13. *The* performance contract *[169] of task N at time instance* $start(N) \leq t < end(N)$ *is defined as:*

$$PC(N, \mathcal{S}_N, t) = \frac{W_N}{W_N(t) \cdot T_N^{\mathcal{S}_N}} \cdot (t - start(N)),$$

where $W_N(t)$ *is the work completed by task N in the time interval* $[start(N), t]$. *The task N is migrated at time instance t iff:*

$$PC(N, \mathcal{S}_N, t) > f_N,$$

where f_N *is the* performance contract elapse factor *of task N.*

Each task has a statically associated a performance contract elapse factor f_N (as part of the workflow specification) that represents a certain percentage from the predicted task execution time $T_N^{\mathcal{S}_N}$.

There are two options for computing the work $W_N(t)$:

1. through sensors hard-coded within each individual task. This approach has the advantage of being precise, but requires code instrumentation (i.e., either the source code, or the binary code based on the dynamic instrumentation technology described in Section 5.2.1);
2. through standard online metrics (e.g., hardware counters [22]) provided by the Dynamic Instrumentor service described in Section 5.3.7.

Upon migration, the workflow tasks are restarted (restart based on checkpointing [109, 146] will be considered in future work).

6.3.2 Static DAG Generation

Let $(Nodes, Edges)$ denote a statically scheduled DG-based workflow running at the time instance t. The dynamic scheduling algorithm outlined in Figure 6.6 is based on iterative invocations of the (genetic) static scheduling algorithm. The *static DAG* $\mathcal{A}_t = (Nodes_t, Edges_t)$ given as input to the static scheduler at scheduling event t is constructed using the following rules:

- $Nodes_t$ comprises the executing tasks $Nodes_{Migr}$ that require migration and the tasks $Nodes_{Queued}$ which are queued (properly running tasks like N_3 in Figure 6.7 are eliminated):

$$Nodes_t = Nodes_{Migr} \cup Nodes_{Queued},$$

where:

- $Nodes_{Migr}$ comprises the executing tasks that require migration due to failures or performance contract violation:

$$Nodes_{Migr} = \{N \cup Nodes_{PC}^{N} \mid \forall\, N \in Nodes \wedge state(N,t) = failed \vee (state(N,t) = running \wedge PC(N, \mathcal{S}_N, t) > f_N)\}$$

(e.g., see the tasks FT_4 and JS_5 in Figure 6.7, assuming that the task JS_5 violated its performance contract), where:

- $Nodes_{PC}^{N}$ comprises the tasks already completed which need to be resubmitted upon the migration of the task $N = JS(z) \in Nodes^{JS} \vee N = FT(z, z') \in Nodes^{FT}$ due to the static schedule dependencies induced by the machine z (the static schedule dependencies have been defined in Section 6.2.2):

$$Nodes_{PC}^{N} = \left\{JS'(z) \in pred^{p}(N)\right\} \cup \left\{FT'(z', z) \in pred^{p}(N)\right\}$$

(e.g., see the task $FT_4(z_1, z_2)$ in Figure 6.7 which contains z_2 as a static schedule dependency to the task $JS_5(z_2)$);

- $Nodes_{Queued}$ comprises the tasks which are queued and have not yet been submitted:

$$Nodes_{Queued} = \{N \in succ^{p}(N_s) \mid \forall\, N_s \in Nodes \wedge state(N_s) = running\}.$$

The completed tasks which are part of workflow loops are therefore included for the next iteration (e.g., see the tasks N_1, N_2, FT_4, N_6 in Figure 6.7);

- $Edges_t$ comprises the edges that connect the subworkflow tasks from $Nodes_t$ and *eliminate the workflow cycles*:

$$Edges_t = Edges_{DAG} - Edges_{Migr} - Edges_{Queued},$$

where:

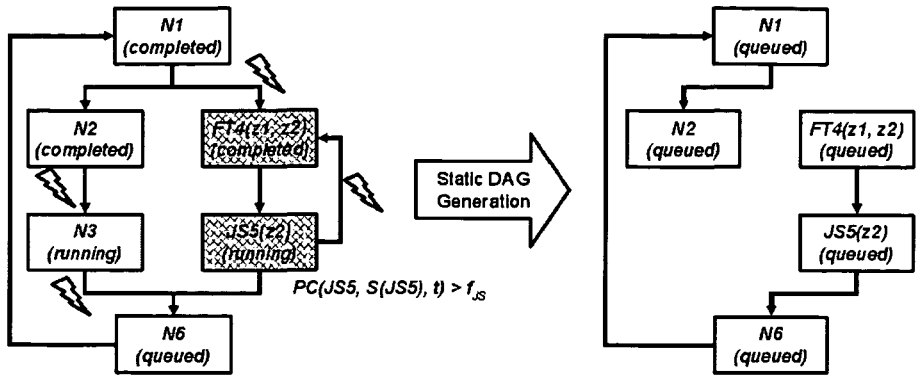


Fig. 6.7. Sample static DAG generation, where the task JS_5 violates the performance contract.

- $Edges_{DAG}$ comprises the entire subset of edges that connect the workflow tasks from $Nodes_t$:

$$Edges_{DAG} = \{(N, N') \in Edges \mid \forall N \in Nodes_t \wedge \forall N' \in Nodes_t\}.$$

  The subset $Edges_{DAG}$ eliminates the edges which contain properly running tasks that fulfil their performance contract (e.g., see the edges (N_2, N_3) and (N_3, N_6) within the loop $(N_1, N_2, N_3, N_6, N_1)$ in Figure 6.7);
- $Edges_{Migr}$ eliminates the cycles within the loops which contain tasks that require migration (e.g., see the edge (N_1, FT_4) within the loop $(N_1, FT_4, JS_5, N_6, N_1)$ in Figure 6.7):

$$Edges_{Migr} = \{(N, N') \in Edges \mid \forall N \in Nodes_{Queued} \wedge \forall N' \in Nodes_{Migr}\};$$

- $Edges_{Queued}$ breaks the remaining cyclic execution paths ρ by eliminating the edges that violate the node topological order (e.g., see the edge (JS_5, FT_4) within the loop (FT_4, JS_5, FT_4) in Figure 6.7):

$$Edges_{Queued} = \{(N_p, N_1) \mid \forall \rho = (N_1, \ldots, N_p, N_1)\}.$$

6.4 Static Throughput Scheduling

Scheduling multiple (independent or parameter study) applications (referred in the following as *tasks*) for high performance throughput is an important optimisation problem on the Grid, which is well known as NP-complete [90].

The scope of this section is to illustrate an instantiation the ZENTURIO optimisation framework for throughput scheduling of independent tasks. The problem is approached as a specialisation of the static workflow scheduling approach presented in Section 6.2 using the model introduced in Section 2.8.3.

Providing the appropriate optimisation (fitness) function relies on providing appropriate prediction models for each independent task, for instance as introduced in Definition 6.9.

Example 6.14 (Java independent task-set).

```
//ZEN$ SUBSTITUTE z1 = { e{1:100} }
//ZEN$ SUBSTITUTE z2 = { e{1:100} }
//ZEN$ SUBSTITUTE z3 = { e{1:100} }
//ZEN$ SUBSTITUTE z4 = { e{1:100} }
//ZEN$ SUBSTITUTE z5 = { e{1:100} }

Task N1 = createJS(''z1'');
Task N2 = createJS(''z2'');
Task N3 = createJS(''z3'');
Task N4 = createJS(''z4'');
Task N5 = createJS(''z5'');

TaskGraph taskSet = new TaskGraphImpl();
taskGraph.add(N1);
taskGraph.add(N2);
taskGraph.add(N3);
taskGraph.add(N4);
taskGraph.add(N5);

Dependency dependency = new DependencyImpl();
taskSet.setDependency(dependency);
```

Example 6.14 defines a set of five independent task as a ZEN application $\mathcal{A}(z_1, z_2, z_3, z_4, z_5)$, where each ZEN variable z_i represents the abstract machine that hosts the task N_i, $\forall\, i \in \{1, 2, 3, 4, 5\}$. There are no schedule dependencies between the tasks. The ZEN directives define the set of possible concrete instantiations (with cardinality 100) of each abstract machine.

The following definition specifies the generic instantiation of the ZENTURIO optimisation framework and the genetic search engine for the static throughput scheduling problem.

Definition 6.15. *A* ZEN application *is an aggregation of n independent tasks:* $\mathcal{A}(z_1, \ldots, z_n) = (Nodes = \{JS_1(z_1), \ldots, JS_n(z_n)\}, \phi)$. *A* ZEN variable (gene) z_i *is a parameter that represents an abstract Grid machine where the task* JS_i *executes. The* value set $\mathcal{V}^{z_i}$ *of a ZEN variable* z_i *represents the entire set of concrete Grid machines. An* allele $e_i \in \mathcal{V}^{z_i}$ *is a concrete machine of the Grid. A* task schedule *is a function that maps each task onto a concrete machine from the Grid:*

$$\mathcal{S} : Nodes \rightarrow \mathcal{V}^{z_i}.$$

An individual *is a ZEN application instance* $\mathcal{AI}(e_1, \ldots, e_n)$, *where* $\mathcal{S}(N_i) = e_i$, *where* $e_i \in \mathcal{V}^{z_i}$, $\forall\, i \in [1..n]$.

The crossover and mutation operators for independent tasks can be graphically represented as already illustrated in Figure 6.3.

Definition 6.16. *Let $\mathcal{V}^z$ denote the full set of machines in the Grid, N a set of tasks, and $\mathcal{S} : N \rightarrow \mathcal{V}^z$ the task schedule function. The* Gantt chart *of N is a function:*

$$\mathcal{G} : \mathcal{V}^z \rightarrow \mathcal{P}(N),\ \mathcal{G}(e) = \{\{N_1, \ldots, N_p\} \mid \mathcal{S}(N_i) = e,\ \forall\, i \in [1..p]\},$$

where $\mathcal{P}$ denotes the power set. The throughput fitness function *is defined as:*

$$\mathcal{F} : \mathcal{V}^{\mathcal{A}} \rightarrow \mathbb{R}_+,\ \mathcal{F}(\mathcal{AI}(e_1, \ldots, e_n)) = \mathcal{C} - \max_{e \in \{e_1, \ldots, e_n\}} \left(\sum_{i=1}^{p} T^e_{N_i} \right),$$

where $\mathcal{G}(e) = \{N_1, \ldots, N_p\}$ and $\mathcal{C}$ is a constant. Maximising the throughput fitness function is the throughput scheduling problem.

Informally, with each individual there is associated a Gantt chart that maps each task onto one Grid machine. The tasks scheduled on the same machine are executed sequentially in irrelevant order. The machine with the maximum execution time gives the schedule *makespan* that needs to be minimised. The fitness function is defined by the makespan subtracted from a large enough constant $\mathcal{C}$.

The throughput fitness function or the makespan for the five tasks illustrated in Example 6.14 can be expressed as:

$$\mathcal{F}(\mathcal{AI}(e_1, e_2, e_3, e_1, e_1)) = \mathcal{C} - max\left(T^{e_1}_{N_1} + T^{e_1}_{N_4} + T^{e_1}_{N_5}, T^{e_2}_{N_2}, T^{e_3}_{N_3}\right),$$

assuming the following task schedules, also shown in the Gantt chart depicted in Figure 6.8:

$$\begin{aligned} \mathcal{S}(N_1) = \mathcal{S}(N_4) = \mathcal{S}(N_5) &= e_1; \\ \mathcal{S}(N_2) &= e_2; \\ \mathcal{S}(N_3) &= e_3. \end{aligned}$$

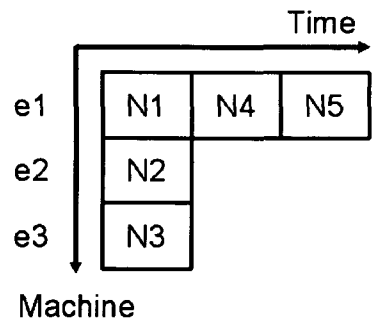


Fig. 6.8. Sample Gantt chart for the task set defined in Example 6.14.

Past community research has already addressed the throughput scheduling problem for parallel and Grid computing and developed a class of good heuristic algorithms like Min-min, Max-min, and (X)Sufferage [112]. All of these algorithms exhibit a complexity of $\mathcal{O}(T^2 \cdot M)$, where T is the number of tasks and M the number of available machines (processors). On large computational Grids that aggregate a potentially unbounded number of resources and run a rather large number of concurrent tasks, this $\mathcal{O}(n^3)$ complexity (where n stands for both unlimited task number and Grid size) is becoming critical.

The genetic algorithm proposed by this section has the advantage of delivering potentially good-enough solutions significantly faster. Assuming a number of tasks n, a population size $p < n$, and a maximum generation number $g < n$, the algorithm exhibits a complexity of $\mathcal{O}(p \cdot g \cdot n)$, where $p \cdot g$ represents the number of visited search points. The experimental results reported in Section 7.3.2 show that converging to good results requires $p \cdot g$ be of the same order of magnitude as the number of tasks, which leads to an $\mathcal{O}(n^2)$ complexity. Obviously, the quality of the solutions delivered will be worse compared to those provided by the previous heuristics. For instance for scheduling a set of 1000 tasks on a Grid consisting of 100 machines, the genetic algorithm obtains solutions from 25% to 50% worse compared to those given by the Max-min heuristic. The scope of the algorithm is, however, for larger problem sizes (i.e., task number and Grid size) for which the Max-min heuristics become impractical due to the large number of schedules to be evaluated.

6.5 Optimisation of Parallel Applications

Finding appropriate parameter combinations, often representing parallelisation options, that optimise a certain performance metric (usually minimise the execution time) is known as *performance tuning*. The objective function for performance tuning of parallel applications using the ZENTURIO optimisation framework is represented by a performance metric (or an arithmetical combination of multiple performance metrics) indicated through ZEN performance directives (see Section 3.8).

Definition 6.17. *Let $\mathcal{A}$ denote a ZEN application, $\mathcal{M}$ a performance measurement as defined in Section 3.8 (see Definition 3.24), and M a target execution machine. The* objective function *for performance tuning of parallel applications is defined as follows:*

$$\mathcal{F} : \mathcal{V}^{\mathcal{A}} \rightarrow \mathbb{R},\ \mathcal{F}(\mathcal{AI}) = \delta_M(\mathcal{M}, \mathcal{AI}),$$

where δ_M is the performance data defined in Definition 3.26.

Let $\mathcal{CR}$ denote the outermost code region of a ZEN application (i.e., the entire application), as introduced in Definition 3.24, and M a target execution parallel machine. In the following, a few sample objective functions

representing useful performance metrics to be tuned using ZENTURIO are formally defined. Since some of the metrics require minimisation, they had to be subtracted from a large-enough constant C.

- *Execution Time:*
$$\mathcal{F}(\mathcal{AI}) = C - \delta_M(\mathcal{M}, \mathcal{AI}),$$
where $\mathcal{M} = (\texttt{WTIME}, \mathcal{CR})$ and WTIME denotes the wallclock time metric (see Definition 3.24);
- *Communication Time:*
$$\mathcal{F}(\mathcal{AI}) = C - \delta_M(\mathcal{M}, \mathcal{AI}),$$
where $\mathcal{M} = (\texttt{COMM}, \mathcal{CR})$ and COMM denotes the communication time metric;
- *Speedup:*
$$\mathcal{F}(\mathcal{AI}(e)) = \frac{\delta_M(\mathcal{M}, \mathcal{AI}(e_0))}{\delta_M(\mathcal{M}, \mathcal{AI}(e))},$$
where $\mathcal{M} = (\texttt{WTIME}, \mathcal{CR})$, z is a ZEN variable that represents the application machine size, $e, e_0 \in \mathcal{V}^z$, and $\mathcal{AI}(e_0)$ represents the sequential version of $\mathcal{A}$;
- *Efficiency:*
$$\mathcal{F}(\mathcal{AI}(e)) = \frac{\delta_M(\mathcal{M}, \mathcal{AI}(e))}{\vartheta^{-1}(e) \cdot \delta_M(\mathcal{AI}(\mathcal{M}, e_0))},$$
where $\mathcal{M} = (\texttt{WTIME}, \mathcal{CR})$, z is a ZEN variable that represents the application machine size, $\vartheta^{-1}(e)$ is the machine size, $e, e_0 \in \mathcal{V}^z$, and $\mathcal{AI}(e_0)$ represents the sequential version of $\mathcal{A}$;
- *Speed:* [151]
$$\mathcal{F}(\mathcal{AI}) = \frac{\delta_M(\mathcal{M}_2, \mathcal{AI})}{\delta_M(\mathcal{M}_1, \mathcal{AI})},$$
where $\mathcal{M}_1 = (\texttt{WTIME}, \mathcal{CR})$, $\mathcal{M}_2 = (\texttt{FPIS}, \mathcal{CR})$, and FPIS denotes the floating point instructions per second metric;
- *Average Speed:* [151]
$$\mathcal{F}(\mathcal{AI}(e)) = \frac{\delta_M(\mathcal{M}_2, \mathcal{AI})}{\delta_M(\mathcal{M}_1, \mathcal{AI}) \cdot \vartheta^{-1}(e)},$$
where $\mathcal{M}_1 = (\texttt{WTIME}, \mathcal{CR})$, $\mathcal{M}_2 = (\texttt{FPIS}, \mathcal{CR})$, z is a ZEN variable that represents the machine size of $\mathcal{A}$, $e \in \mathcal{V}^z$, and $\vartheta^{-1}(e)$ is the machine size;
- *Scalability:* [151]
$$\mathcal{F}(\mathcal{AI}(e_1, e_2)) = \frac{\vartheta^{-1}(e_1', e_2') \cdot \delta_M(\mathcal{M}, \mathcal{AI})}{\vartheta^{-1}(e_1, e_2) \cdot \delta_M(\mathcal{M}, \mathcal{AI}')},$$
where $\mathcal{M} = (\texttt{FPIS}, \mathcal{CR})$, z_1 and z_2 are ZEN variables that represent the problem size, respectively the machine size of $\mathcal{A}$, $e_2, e_2' \in \mathcal{V}^{z_2}$, $\vartheta^{-1}(e_2)$ and $\vartheta^{-1}(e_2')$ are the machine sizes of $\mathcal{AI}$ and $\mathcal{AI}'$, and $\mathcal{AI}'(e_1', e_2')$ is a reference problem-machine size.

6.5.1 MPI Grid Applications

Even though the tightly-coupled parallel programming paradigm introduced in Section 2.8.1 contradicts with the loosely-coupled Grid model, it has been employed for gaining initial experiences on executing existing parallel applications on the Grid. The MPICH-G library extends the MPICH [79] modular design with a new `globus` communication device that enables transparent GridFTP-based communication between MPI processes running on different Grid sites, while using a local optimised (potentially native) MPI installation for communication between local processes. The MPI application is submitted to multiple Grid sites using the DUROC [39] co-allocator provided by the Globus toolkit. MPICH-G [63] enables straight-forward transparent porting of existing parallel MPI applications on the Grid by simply relinking the compiled parallel application.

Optimising MPI applications for a heterogeneous set of Grid resources raises complex *load balancing* problems which are difficult to meet due to the low level of abstraction of the message passing paradigm (i.e., often called fragmented programming).

6.5.2 High Performance Fortran on the Grid

High Performance Fortran (HPF) [84] allows to express array distributions at a high-level of abstraction, while offering the programmer a single program view which is not fragmented by low-level message passing library routines. Special purpose HPF compilers, like the Vienna Fortran Compiler [15] are used to translate a high-level HPF application into an MPI equivalent.

This section proposes a case study on applying the ZENTURIO optimisation framework for static scheduling of ZEN applications containing one single irregularly distributed two-dimensional HPF array. The complex load balancing issues raised by the highly heterogeneous Grid infrastructure can be effectively addressed through the HPF *irregular distributions.*

General Block Distribution

Let $\texttt{MAT}(m, n)$ denote a two-dimensional matrix and $\texttt{PROC}(p, q)$ a two-dimensional processor array.

Definition 6.18. *Let $Bx(p)$ and $By(q)$ denote two one-dimensional distribution arrays, such that: $\sum_{i=1}^{p} Bx_i \geq m$ and $\sum_{i=1}^{q} By_i \geq n$. The* general block data distribution *of MAT is a function:*

$$DISTR : [1..m] \times [1..n] \rightarrow [1..p] \times [1..q],\ DISTR(x, y) = (z, w),$$

where:

$$\sum_{i=1}^{z-1} Bx_i < x \leq \sum_{i=1}^{z} Bx_i, \ \forall\, x \in [1..p];$$
$$\sum_{i=1}^{w-1} By_i < y \leq \sum_{i=1}^{w} By_i, \ \forall\, y \in [1..q].$$

Each distribution array element $Bx_i, \forall\ i \in [1..p]$ *and* $By_j, \forall\ j \in [1..q]$ *is a* ZEN variable *annotated to specify the complete set of possible general block distributions.*

Example 6.19 (HPF general block array distribution).

```
      INTEGER, PARAMETER m = 4
      INTEGER, PARAMETER n = 8
      INTEGER, PARAMETER p = 2
      INTEGER, PARAMETER q = 3
      REAL MAT(m, n)
!HPF$ PROCESSOR PROC(p, q)
      INTEGER, PARAMETER :: x1 = 3
      INTEGER, PARAMETER :: x2 = 1
      INTEGER, PARAMETER :: y1 = 2
      INTEGER, PARAMETER :: y2 = 2
      INTEGER, PARAMETER :: y3 = 4
!ZEN$ SUBSTITUTE x1 = { 0 : m } BEGIN
!ZEN$ SUBSTITUTE x2 = { 0 : m } BEGIN
!ZEN$ SUBSTITUTE y1 = { 0 : n } BEGIN
!ZEN$ SUBSTITUTE y2 = { 0 : n } BEGIN
!ZEN$ SUBSTITUTE y3 = { 0 : n } BEGIN
      INTEGER, PARAMETER :: Bx(p) = (/ x1, x2 /)
      INTEGER, PARAMETER :: By(q) = (/ y1, y2, y3 /)
!ZEN$ END SUBSTITUTE
. . .
!HPF$ DISTRIBUTE MAT(GEN_BLOCK(Bx), GEN_BLOCK(By)) ONTO PROC
!ZEN$ CONSTRAINT VALUE x1 + x2 == 4
!ZEN$ CONSTRAINT VALUE y1 + y2 + y3 == 8
```

Example 6.19 defines the matrix MAT(m, n) which has both dimensions distributed over the processor array PROC(p, q) using the HPF general block mapping arrays $Bx(p)$ and $By(q)$ (see Figure 6.9). The elements of the mapping arrays $Bx(p)$ and $By(q)$ are program constants which are annotated with ZEN substitute directives that specify the complete set of general block distribution possibilities. A distribution of size zero on one processor controls the machine size, since that processor will not take part in the computation. The constraint directives insure that the sum of the general block mapping elements match the matrix rank in each dimension (see Definition 6.18). These

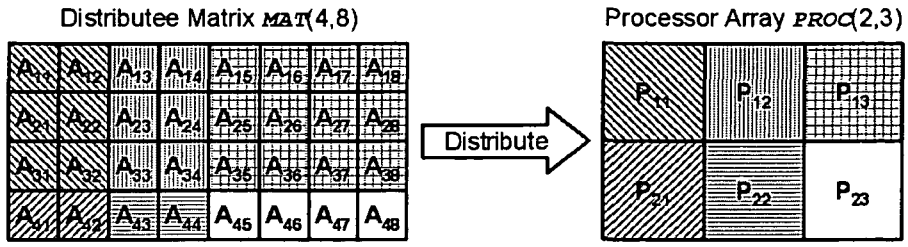


Fig. 6.9. The default general block array distribution defined in Example 6.19.

ZEN annotations define a search space of possible array mappings of size $(m+1)^{p-1} \cdot (n+1)^{q-1}$ (two orders of magnitude are eliminated by the two constraints). The HPF and MPI models (improperly) consider the Grid as a single parallel computer. The HPF PROCESSORS statement in this approach represents the complete set of Grid machines (i.e., of cardinality $p \cdot q$) organised into a two-dimensional array PROC(p, q).

A ZEN application annotated according to the Definition 6.18 (and exemplified in Example 6.19) represents the input to the ZENTURIO optimisation framework. The realisation of the search engine is, e.g., as described in Section 6.1 and requires no further attention. The only missing issue is the instantiation of the objective function, which will be discussed in the following.

Definition 6.20. *Let $\mathcal{A}$ denote a ZEN application (e.g., as sketched in Example 6.19), $\mathcal{CR}$ the outermost code region (i.e., entire program), $\mathcal{M}_1 = (\mathit{COMP}, \mathcal{CR})$ the computation performance measurement, $\mathcal{M}_2 = (\mathit{COMM}, \mathcal{CR})$ the communication performance measurement (see Definition 3.24), and M a parallel machine. Let DISTR denote the (general block) distribution of array MAT. The* objective function *can be approximated as follows:*

$$\mathcal{F}(\mathcal{AI}) = \delta_M(\mathcal{M}_1, \mathcal{AI}) + \delta_M(\mathcal{M}_2, \mathcal{AI}),$$

where:

$$\delta_M(\mathcal{M}_1, \mathcal{AI}) = \max_{\forall i \in [1..p],\ \forall j \in [1..q]} \{\delta_M(\mathcal{M}_1, \mathcal{AI}(\mathit{PROC}_{i,j}, \mathit{DISTR}))\},$$

$$\delta_M(\mathcal{M}_2, \mathcal{AI}) = \max_{\forall i \in [1..p],\ \forall j \in [1..q]} \{\delta_M(\mathcal{M}_2, \mathcal{AI}(\mathit{PROC}_{i,j}, \mathit{DISTR}))\},$$

$\forall\ i \in [1..p],\ \forall\ j \in [1..q]$, *where $\mathcal{AI}(\mathit{PROC}_{i,j}, \mathit{DISTR})$ denotes the partition of $\mathcal{AI}$ hosted by the machine $\mathit{PROC}_{i,j}$ according to the array distribution DISTR.*

For *static scheduling* problems, the computation and the communication performance data, denoted as $\delta_M(\mathcal{M}_1, \mathcal{AI}(\mathtt{PROC}_{i,j}, \mathtt{DISTR}))$ respectively $\delta_M(\mathcal{M}_2, \mathcal{AI}(\mathtt{PROC}_{i,j}, \mathtt{DISTR}))$, could be approximated through application-specific analytical prediction models. For instance, a *Jacobi relaxation* performs the same computation repeatedly on all matrix elements, while the

communication requires exchanging the boundary elements with all the neighbouring processors. In case of the general block array distribution, these can be analytically approximated as follows:

$$\delta_M(\mathcal{M}_1, \mathcal{AI}(\texttt{PROC}_{i,j}, \texttt{DISTR})) = Bx_i \cdot By_j \cdot \frac{W_e}{v_e} \cdot I;$$

$$\begin{aligned}\delta_M(\mathcal{M}_2, \mathcal{AI}(\texttt{PROC}_{i,j}, \texttt{DISTR})) = {} & \overline{\mathcal{L}(\texttt{PROC}_{i,j})} + \\ & + Bx_i \cdot S_e \cdot \left(\frac{1}{\mathcal{B}(\texttt{PROC}_{i,j}, \texttt{PROC}_{i-1,j})} + \frac{1}{\mathcal{B}(\texttt{PROC}_{i,j}, \texttt{PROC}_{i+1,j})} \right) + \\ & + By_j \cdot S_e \cdot \left(\frac{1}{\mathcal{B}(\texttt{PROC}_{i,j}, \texttt{PROC}_{i,j-1})} + \frac{1}{\mathcal{B}(\texttt{PROC}_{i,j}, \texttt{PROC}_{i,j+1})} \right),\end{aligned}$$

$\forall\, i \in [1..p],\ \forall\, j \in [1..q]$, where:

- $W_e = \delta_M(\texttt{FP\_INST}, \mathcal{CR})$ is the work required to compute one matrix element, expressed in floating point instructions;
- v_e is the machine speed that computes the matrix element expressed in floating point instructions per second (e.g., as measured by the LINPACK benchmark [47]);
- I is the number of iterations;
- S_e is the size in bytes of a matrix element, i.e., $S_e = sizeof(e)$;
- $\overline{\mathcal{L}(\texttt{PROC}_{i,j})}$ is the total latency of the communication with the four matrix element neighbours;

$$\begin{aligned}\overline{\mathcal{L}(\texttt{PROC}_{i,j})} = {} & \mathcal{L}(\texttt{PROC}_{i,j}, \texttt{PROC}_{i-1,j}) + \mathcal{L}(\texttt{PROC}_{i,j}, \texttt{PROC}_{i+1,j}) + \\ & + \mathcal{L}(\texttt{PROC}_{i,j}, \texttt{PROC}_{i,j-1}) + \mathcal{L}(\texttt{PROC}_{i,j}, \texttt{PROC}_{i,j+1});\end{aligned}$$

- $\mathcal{L}(\texttt{PROC}_{i,j}, \texttt{PROC}_{k,l})$ is the latency between the processors $\texttt{PROC}_{i,j}$ and $\texttt{PROC}_{k,l}$;
- $\mathcal{B}(\texttt{PROC}_{i,j}, \texttt{PROC}_{k,l})$ is the bandwidth between the processors $\texttt{PROC}_{i,j}$ and $\texttt{PROC}_{k,l}$.

Indirect Distribution

The same technique presented in the previous section can be applied on the more general indirect array distribution.

Let $\texttt{MAT}(m, n)$ denote a two-dimensional matrix and $\texttt{PROC}(p, q)$ a two-dimensional processor array.

Definition 6.21. *Let $I(p, q)$ denote a one-dimensional distribution array, such that $I(i, j) \leq p \cdot q$, $\forall\, i \in [1..p]$, $\forall\, j \in [1..q]$. The* indirect data distribution *of MAT is a function:*

$$DISTR\colon [1..m] \times [1..n] \rightarrow [1..p] \times [1..q],$$

$$DISTR(x, y) = \left(I(x, y) \bmod m,\ \left\lceil \frac{I(x, y)}{m} \right\rceil \right).$$

The partition:

$$MAT_{PROC_{i,j}} = \{MAT_{k,l} \mid DISTR(k,l) = (i,j),\ \forall\ i \in [0..p],\ \forall\ j \in [0..q]\}$$

is called the distribution *of MAT onto the processor $PROC_{ij}$. Each distribution array element $I(x,y)$, $\forall\ x \in [1..p]$, $\forall\ j \in [1..q]$, is a* ZEN variable *annotated to specify the complete set of possible indirect array distributions.*

Example 6.22 defines the matrix $\texttt{MAT}(m,n)$ which has the elements indirectly distributed across the processor array $\texttt{PROC}(p,q)$ according to the mapping array $MAP(m,n)$ (see Figure 6.10). The elements of the mapping array $MAP(m,n)$ are program constants which are annotated with ZEN substitute directives that specify the complete set of possible indirect distributions. These ZEN annotations define a search space of possible array mappings of size $(m \cdot n)^{pq}$. The HPF PROCESSORS statement in this approach represents the complete set of Grid machines (i.e., of cardinality $p \cdot q$) organised into a two-dimensional array $\texttt{PROC}(p,q)$.

The objective function can be equally expressed as in the context of the general block distribution (see Definition 6.20).

For the Jacobi relaxation application with an irregular array distribution, the computation and the communication performance data could be approximated as follows:

$$\delta_M(\mathcal{M}_1, \mathcal{AI}(\texttt{PROC}_{i,j}, \texttt{DISTR})) = \left|\texttt{MAT}_{\texttt{PROC}_{i,j}}\right| \cdot \frac{W_e}{v_e};$$

$$\delta_M(\mathcal{M}_2, \mathcal{AI}(\texttt{PROC}_{i,j}, \texttt{DISTR})) = \frac{\sum_{i=1}^{p}\sum_{j=1}^{q}\sum_{k=1}^{p}\sum_{l=1}^{q}(\mathcal{L}(\texttt{PROC}_{i,j}, \texttt{PROC}_{k,l}))}{p \cdot q} +$$
$$+ \frac{\sum_{i=1}^{p}\sum_{j=1}^{q}\sum_{k=1}^{p}\sum_{l=1}^{q}(\mathcal{B}(\texttt{PROC}_{i,j}, \texttt{PROC}_{k,l}))}{p \cdot q},$$

where $\left|\texttt{MAT}_{\texttt{PROC}_{i,j}}\right|$ is the cardinality of the distribution of MAT onto the processor $\texttt{PROC}_{i,j}$.

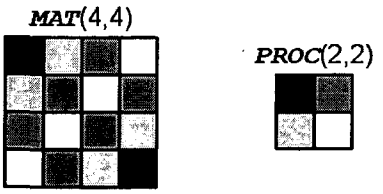


Fig. 6.10. The default indirect array distribution defined in Example 6.22.

Example 6.22 (HPF indirect array distribution).

```
      INTEGER, PARAMETER m = 4
      INTEGER, PARAMETER n = 4
      INTEGER, PARAMETER p = 2
      INTEGER, PARAMETER q = 2
      DIMENSION MAT(m,n)
!HPF$ PROCESSORS PROC(p,q)
      INTEGER, PARAMETER M11 = 1
      INTEGER, PARAMETER M12 = 2
      INTEGER, PARAMETER M13 = 3
      INTEGER, PARAMETER M14 = 4
      INTEGER, PARAMETER M21 = 2
      INTEGER, PARAMETER M22 = 3
      INTEGER, PARAMETER M23 = 4
      INTEGER, PARAMETER M24 = 2
      INTEGER, PARAMETER M31 = 3
      INTEGER, PARAMETER M32 = 4
      INTEGER, PARAMETER M33 = 3
      INTEGER, PARAMETER M34 = 2
      INTEGER, PARAMETER M41 = 4
      INTEGER, PARAMETER M42 = 3
      INTEGER, PARAMETER M43 = 2
      INTEGER, PARAMETER M44 = 1
!ZEN$ SUBSTITUTE M11 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M12 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M13 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M14 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M21 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M22 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M23 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M24 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M31 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M32 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M33 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M34 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M41 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M42 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M43 = { 1 : 4 } BEGIN
!ZEN$ SUBSTITUTE M44 = { 1 : 4 } BEGIN
      INTEGER MAP(m,n) = (/ (/ M11, M12, M13, M14 /),
                            (/ M21, M22, M23, M24 /),
                            (/ M31, M32, M33, M34 /),
                            (/ M41, M42, M43, M44 /) /)
!ZEN$ END SUBSTITUTE
```

```
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!ZEN$ END SUBSTITUTE
!HPF$ DISTRIBUTE MAT(INDIRECT(MAP)) ONTO PROC
```

7

Experiments

This chapter presents a variety of experiments that use the ZENTURIO experiment management tool for:

1. *performance studies* of several real-world OpenMP and MPI parallel applications, including:
   a) an ocean simulation (see Section 7.1.1);
   b) a material science kernel (see Section 7.1.2);
   c) a photonic application (see Section 7.1.3);
   d) a benders decomposition method of a financial application (see Section 7.1.4);
   e) two three-dimensional Fast Fourier Transform kernels (see Section 7.1.5);
   f) two Grid services (see Sections 7.1.6 and 7.1.7);
2. *parameter study* of a financial application (see Section 7.2.1);
3. *scheduling Grid applications* as an optimisation and dynamic steering problem, in particular:
   a) a material science workflow application (see Section 7.3.1);
   b) an independent task simulator (see Section 7.3.2).

7.1 Performance Studies

This section describes a set of cross-experiment performance studies using the ZENTURIO experiment management tool on a variety of scientific parallel applications.

The application parameters and the performance metrics of interest are specified using the ZEN directive-based language specified in Chapter 3. The ZENTURIO experiment management tool has been used to automatically generate and conduct all the complete set of experiments and store the output results and the performance data into the Experiment Data Repository, as presented in Chapter 4. The Application Data Visualiser portlet of the user portal introduced in Section 4.1.4 has been used to automatically formulate

SQL queries and generate post-mortem visualisation diagrams that display the variation of any (set of) performance data as a function of arbitrary application parameters (i.e., ZEN variables).

Unless differently stated, the experiments have been conducted on an SMP (Symmetric Multiprocessor) cluster that consists of 16 four-way Intel Pentium III Xeon 700 MHz processors interconnected through both Fast Ethernet and Myrinet network cards. The experiments have been submitted for execution on the dedicated cluster nodes using GRAM [38] as the job manager and PBS [166] as the back-end job scheduler.

7.1.1 Ocean Simulation

The *Stommel* model [150] has been thought with the purpose of explaining the westward intensification of wind-driven ocean currents. This section presents a performance study of a mixed OpenMP and MPI parallel Fortran90 implementation of the Stommel model.

The following parameters have been specified for this application through ZEN directives:

1. *The machine size* consists of two dimensions:
   a) *The number of threads per SMP node* are controlled by the `NUM_THREADS` clause of the OpenMP `PARALLEL` directive (see Example 7.1);
   b) *The number of SMP nodes* are controlled through directives inserted in the Globus RSL script as illustrated in Example 7.2.

   Each MPI experiment has been submitted as a `single` GRAM job type which allows to choose between various local communication libraries. The `count` parameter in Example 7.2 is assigned a value equal to the number of SMP nodes times the number of processors per node, which was necessary for persuading GRAM to allocate the correct number of nodes. The shell script `script.sh` used to start the MPI application (see Example 7.3) sets the maximum number of MPI processes per node to one through the `MPI_MAX_CLUSTER_SIZE` environment variable. One single MPI process per SMP node leaves the intra-node parallelisation to the OpenMP compiler;
2. *Two interconnection networks* (i.e., Fast Ethernet and Myrinet) have been examined by linking the application with the corresponding MPICH library. The MPI library implementations are indicated by annotating the `MPILIB` variable in the application `Makefile`, as shown in Example 7.4. The constraint directive makes the correct association between the implementation specific MPI libraries and external `MPIRUN` ZEN variable (defined in Example 7.3) which contains the path to the `mpirun` script that starts the application;
3. *The problem size* has been varied by changing the grid (ocean) size and the number of iterations, as shown in Example 7.5;
4. *The performance metrics* of interest for every experiment are the execution time and the communication overhead (i.e., the mnemonics `WTIME`

and ODATA), which have been measured for the entire program and the outermost OpenMP loop (i.e., the mnemonics CR_P and CR_OMPPA), as shown by the ZEN performance directive in Example 7.1.

Example 7.1 (Source code excerpt).

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

Example 7.2 (Globus RSL script).

```
(*ZEN$ SUBSTITUTE count\=4 = { count={1:10} }*)
& (count=4)
  (jobtype=single)
  (directory="/home/radu/APPS/STOMMEL_OMPI")
  (executable="script.sh")
  (stdin="st.in")
  (stdout="st.out")
```

Example 7.3 (Shell script – `script.sh`).

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

Example 7.4 (`Makefile`).

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

Example 7.5 (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 nine ZEN directives have been included in three files of this application to specify a total of 160 experiments:

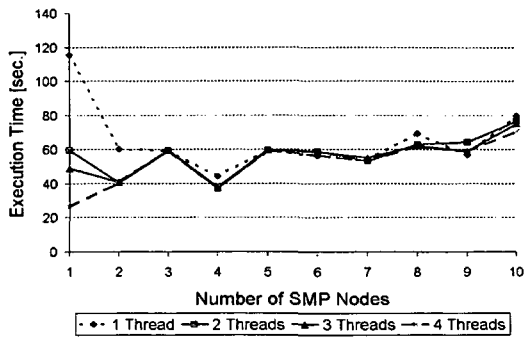
$$|\mathcal{V}(nodes{=}2, count{=}4, \texttt{MPIRUN}, \texttt{NUM\_THREADS}(4), \texttt{MPILIB}, points, iters)| = 160.$$

For a 200 × 200 problem size, the application does not scale (see Figure 7.1(a)) which is explained by the excessive MPI communication (see Figure 7.1(b)). This problem size, however, scales well with the number of threads on a single SMP node. For larger number of nodes, the number of threads does not influence the overall performance due to the large MPI communication overhead that dominates the intra-node computation parallelised using OpenMP. The same problem size scales much better over Myrinet (see Figure 7.1(c)) which is due to the lower communication cost on the much faster Myrinet interconnection network (compared to Fast Ethernet).

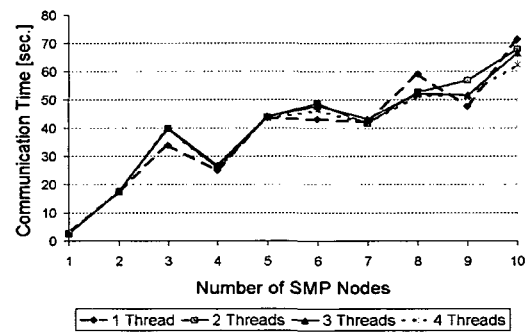
The 400 × 400 problem size shows a very reasonable scaling behaviour until four SMP nodes (see Figure 7.2(a)). Using more than four SMP nodes no longer decreases the execution time substantially. The reason is the increased communication overhead and a decreasing ratio between the computation and the communication times (see Figure 7.2(b)). For smaller number of nodes, the computation to communication time ratio is high and, therefore, the intra-node OpenMP parallelisation yields a satisfactory scaling behaviour. Increasing the number of threads decreases the execution time as expected. Similarly, this problem size scales well over the Myrinet network (see Figure 7.2(c)).

A second experiment was elaborated to show the number of nodes which produce the lowest execution time for different problem sizes over Fast Ethernet (see Figure 7.3(a)). The machine and the problem sizes have been annotated as shown in the Examples 7.2 and 7.5. Employing four OpenMP parallel threads per node yields the best performance for all experiments. The optimal number of SMP nodes increases, as expected, with the problem size. The flat parts of the curve are caused by load balancing problems on odd processor counts due to an uneven array distribution.

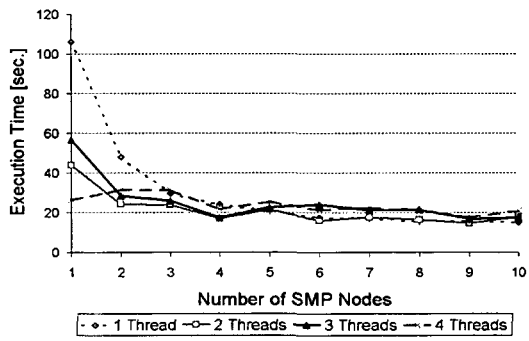
A third experiment was conducted to examine the different OpenMP loop scheduling strategies and their performance effects. The scheduling strategy and the chunk size have been varied using a ZEN substitute directive, as illustrated in Example 3.12 (see Section 3.4.2). The execution time of the OpenMP `PARALLEL` region has been requested through one ZEN performance behaviour directive as shown in Example 7.1. Figure 7.3(b) illustrates that for the problem size examined, the `STATIC` scheduling performs better than the `DYNAMIC` and the `GUIDED` strategies. The optimal chunk size is 50. Static scheduling is superior because it implies the least runtime scheduling overhead.



(a) Fast Ethernet network.

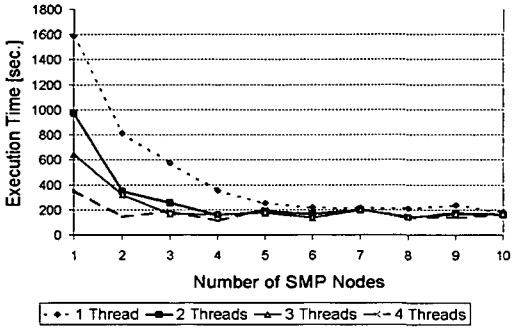


(b) Fast Ethernet network.

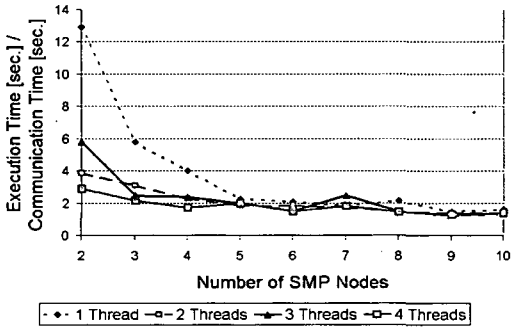


(c) Myrinet network.

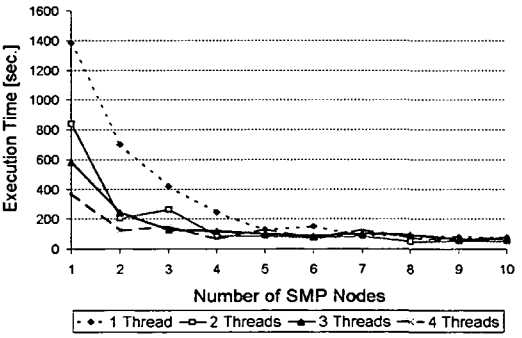
Fig. 7.1. The Stommel model performance results for various intra-node and inter-node machine sizes (I), 200 × 200 problem size, 20000 iterations.



(a) Fast Ethernet network.

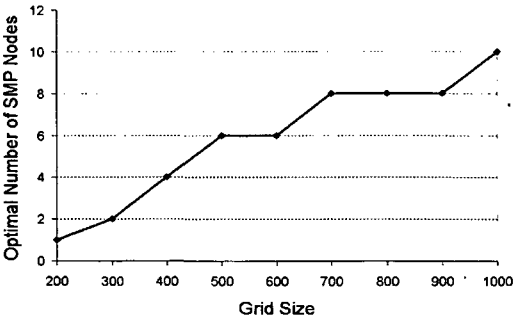


(b) Fast Ethernet network.

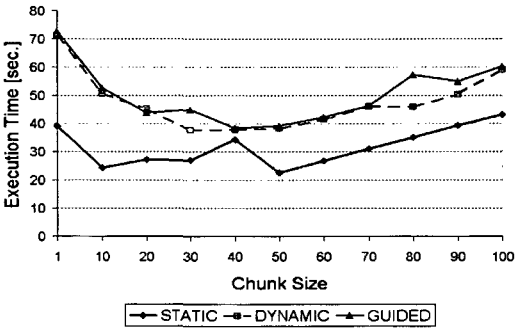


(c) Myrinet network.

Fig. 7.2. The Stommel model performance results for various intra-node and inter-node machine sizes (II), 400 × 400 problem size, 40000 iterations.



(a) Various problem sizes, four threads, 20000 iterations.



(b) Various loop parallelisations, 200 × 200 problem size, 20000 iterations.

Fig. 7.3. The Stommel model performance results (III).

7.1.2 LAPW0

LAPW0 is a material science kernel, part of the Wien2k package [20], that calculates the potential of the Kohn-Sham eigen-value problem. This section presents a performance study of a Fortran90 MPI implementation of LAPW0.

Several application parameters have been varied by means of ZEN directives.

1. *The problem size* is expressed by pairs of `.clmsum` and `.struct` input files, indicated in the `lapw0.def` input file (see Example 7.6). The ZEN substitute directive is used to specify the file locations to the problem sizes of interest, which correspond to 8, 16, 32, and 64 atoms;

2. *The machine size* is controlled by the `nodes=1` and `no_procs` ZEN variables in the PBS script (see Example 7.7) used to submit the experiments on the cluster. This performance study uses ZENTURIO in cluster mode which bypasses GRAM. The ZEN variable `nodes=1` controls the number of SMP nodes and `no_procs` indicates the number of MPI processes to execute. Each node is filled with four MPI processes before allocating a new node. The constraint directive ensures that the correct amount of SMP nodes is allocated for each number of MPI processes. The PBS script also assigns the path of the `mpirun` command to the `MPIRUN` environment variable through a ZEN assignment directive;
3. *The interconnection network* is varied by annotating the `MPILIB` environment variable that specifies the path to the (Fast Ethernet or Myrinet) MPI library in the Makefile used to build the application (see Example 7.8). Shared memory has been used to communicate inside the SMP nodes in case of the Fast Ethernet network. The ZEN constraint directive ensures the correct association between the network specific MPI libraries and the corresponding `mpirun` script;
4. *The performance metrics* measured are the execution time (i.e., the mnemonic `WTIME`) and the communication time (i.e., the mnemonic `ODATA`) for the entire program (i.e., the mnemonic `CR_P`). This is expressed by the ZEN performance directive from Example 7.9.

Example 7.6 (Input data file – `lapw0.def`).

```
!ZEN$ SUBSTITUTE .125hour = {.125hour, .25hour, .5hour, 1hour}
8,'ktp_.125hour.clmsum','old','formatted',0
. . .
20,'ktp_.125hour.struct','old','formatted',0
```

Example 7.7 (PBS script – `run.pbs`).

```
#ZEN$ SUBSTITUTE nodes\=1 = { nodes={1:40} }
#PBS -l walltime=0:29:00,nodes=1:fourproc:ppn=4
cd $PBS_O_WORKDIR
#ZEN$ ASSIGN MPIRUN = { /opt/local/mpich/bin/mpirun,
                        /opt/local/mpich_gm/bin/mpirun.ch_gm }
no_procs = 16
#ZEN$ ASSIGN no_procs = { 1:40 }
$(MPIRUN) -np $no_procs ../SRC/lapw0 lapw0.def
#ZEN$ CONSTRAINT INDEX 4 * (nodes\=1 -1) < no_procs &&
                       no_procs <= 4*nodes\=1 && no_procs != 1
```

Example 7.8 (`Makefile`).

```
#ZEN$ ASSIGN MPILIB = { /opt/local/mpich/lib,
                        /opt/local/mpich_gm/lib }
```

```
#ZEN$ CONSTRAINT INDEX MPILIB == run.pbs:MPIRUN
LIBS = ... -lsismpiwrapper -L$(MPILIB) -lmpich
. . .
$(EXEC): $(OBJS)
        $(F90) -o lapw0 $(OBJS) $(LIBS)
```

Example 7.9 (Fortran source file excerpt - `lapw0.F`).

```
!ZEN$ CR PMETRIC WTIME, ODATA
. . .
```

Eight ZEN directives have been inserted into four ZEN files, based on which a total of 320 experiments were automatically generated and executed by ZENTURIO. Figure 7.4(a) shows the scalability of the application for all the four problems sizes examined. The scalability of the algorithm improves by increasing the LAPW0 problem size (number of atoms). For a problem size of 8 atoms (i.e., `.125hour`) LAPW0 does not scale, which is partially due to the extensive communication overhead with respect to the entire execution time. Figure 7.4(c) shows the contribution of each computed overhead to the overall execution time of each experiment. The unidentified overhead could not be separated from the optimal execution time (i.e., sequential time divided by the number of processes) because a sequential implementation of LAPW0 could not be run due to physical memory limitations. For 64 atoms (i.e., `1hour`), the application scales well up to 16 processes, after which the execution time becomes relatively constant.

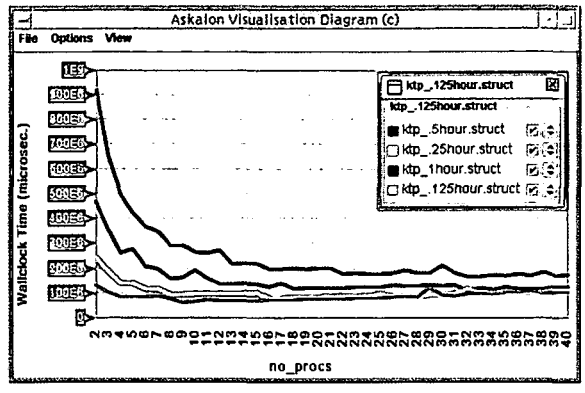
The interconnection network does not improve the communication behaviour (see Figure 7.4(b)) because the blocking time of all the message receive operations dominates the effective transfer of the relatively small amount of data across the processes.

7.1.3 Three-Dimensional Particle-In-Cell

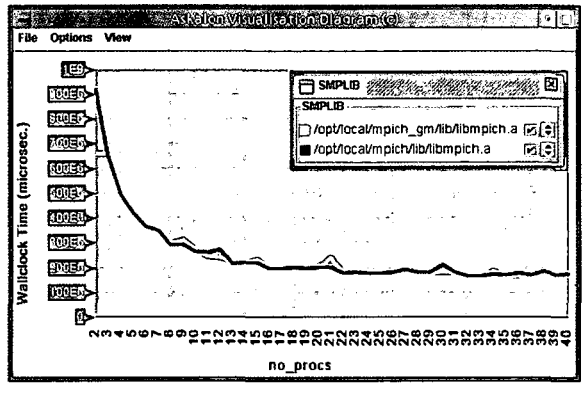
The *three-dimensional Particle-In-Cell (3DPIC)* [75] application simulates the interaction of high intensity ultrashot laser pulses with plasma in three-dimensional geometry. This section presents a performance study of a Fortran90 MPI application of 3DPIC.

The following annotations have been performed.

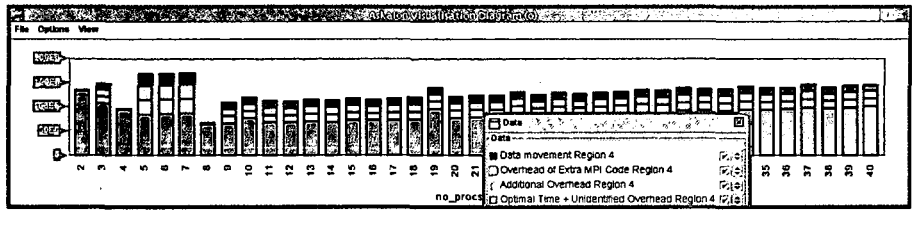
1. *The machine size* is restricted to 1, 4, 9, 12, 16, 25, and 36 parallel processes due to application encoding peculiarities. These have been expressed by the `count` argument of the GRAM RSL script shown in Example 7.10. Based on the number of processes of one experiment, GRAM allocates the correct number of dedicated SMP nodes using PBS. The job type has been set to `single` which allows flexibility in choosing the local interconnection network. The application is started using the shell script illustrated in Example 7.11, which assigns to the `MPIRUN` ZEN variable the path to the `mpirun` script;



(a) Four problem sizes, Fast Ethernet network.



(b) Network comparison (Fast Ethernet versus Myrinet), 64 atoms problem size.



(c) Contribution of the Myrinet communication overhead to the wallclock time, 8 atoms problem size.

Fig. 7.4. LAPW0 performance results for various machine sizes.

2. *The interconnection network* is studied by annotating the application `Makefile` as already shown in Example 7.8 (see Section 7.1.2). Similarly, a constraint directive associates the `mpirun` command with the correct MPI library.
3. *The performance metrics* of interest are the execution time and the communication overhead, which are specified as already shown in Example 7.9 (see Section 7.1.2).

Example 7.10 (Globus RSL script – `run.rsl`).

```
(*ZEN$ SUBSTITUTE count\=4 = { count={1,1,3,3,4,7,9} }*)
& (count=4)
  (jobtype=single)
  (directory="/home/radu/APPS/LAPW0/znse_6")
  (executable="script.sh") )
```

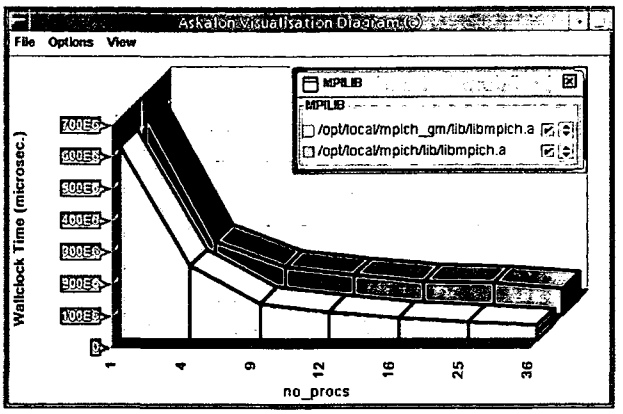
Example 7.11 (Shell script – `script.sh`).

```
#!/bin/sh
cd $PBS_O_WORKDIR
n = `wc -l < $PBS_NODEFILE`
#ZEN$ ASSIGN MPIRUN ={ /opt/local/mpich/bin/mpirun,
                       /opt/local/mpich_gm/bin/mpirun.ch_gm }
$(MPIRUN) -np $n -machinefile $PBS_NODEFILE lapw0
```

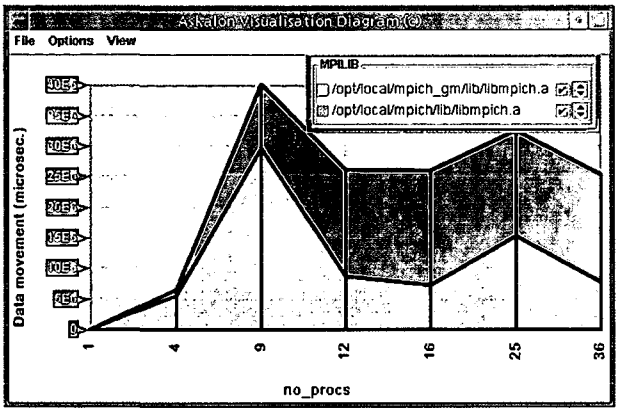
Five ZEN directives have been inserted into four files to generate a total of fourteen experiments. Figure 7.5(a) indicates a good scalability behaviour of the 3DPIC application. The use of the Myrinet network yields about 50% better performance compared to the Fast Ethernet, which is explained by the reduced communication overhead (see Figure 7.5(b)). Figure 7.5(c) shows a relatively low ratio between the application execution time (i.e., one full pie) and the MPI overheads measured, which explains the good application scalability. As a sequential version of this application was not available, the unidentified overhead could not be separated from the optimal execution time.

7.1.4 Benders Decomposition

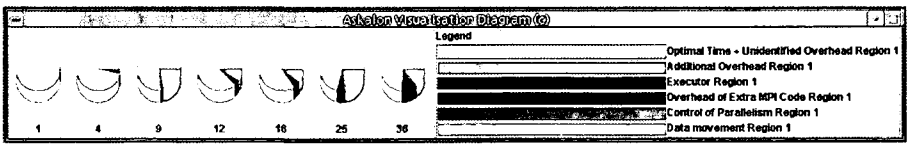
Benders decomposition is a method for structured optimisation, including stochastic optimisation. This section addresses a performance study of a parallel HPF+ [14] implementation of the benders decomposition method employed in the context of a financial application (see Section 7.2.1). HPF+ directives are used to distribute the data across the SMP nodes of the cluster. The HPF+ application is compiled into a hybrid OpenMP and MPI parallel program using the SCALEA [161] instrumentation engine built on top of the



(a) Network comparison (Fast Ethernet versus Myrinet).



(b) Communication overhead comparison (Fast Ethernet versus Myrinet).



(c) Contribution of the Myrinet communication overheads to the wallclock time.

Fig. 7.5. 3DPIC performance results for various machine sizes.

HPF+ Vienna Fortran Compiler [15]. The translated program achieves intra-node parallelisation through OpenMP directives and communication across the SMP nodes through MPI calls.

The following parameters have been studied for this kernel through ZEN directive annotation:

1. *The machine size* consists of two dimensions:
   a) *The number of SMP nodes* is varied by the `count=4` ZEN variable in the Globus RSL script (see Example 7.12). Based on the `count` RSL parameter, GRAM allocates the corresponding number of nodes and uses an available local MPI implementation, which must be defined by the user default shell environment. This experiment uses MPICH on top of the `p4` communication device over Fast Ethernet. The `MPI_MAX_CLUSTER_SIZE` environment variable ensures that the `mpirun` script starts only one MPI process per SMP node, which leaves the intra-node parallelisation to the OpenMP compiler;
   b) *The number of threads per SMP node* is controlled by annotating a global configuration file (see Example 7.13). This information is used by the application in the OpenMP Version 1 parallelisation that does not employ the `NUM_THREADS` clause of a `PARALLEL` region, available only with the version 2 of the standard. This is an example of flexibility which shows how ZENTURIO deals with less elegant or outdated coding styles that does not constrain the non-expert users to learn state-of-the-art programming or adapt the code to the newest specifications.
2. *The performance metrics* of interest for this algorithm are the execution time, the MPI communication time, and the HPF+ inspector and executor overheads [14], which were indicated using one ZEN performance directive similar to the Example 7.9 (see Section 7.1.2).

Example 7.12 (Globus RSL script - `run.rsl`).

```
(*ZEN$ SUBSTITUTE count\=4 = {count={1:10}}*)
& (count=4)
  (jobtype=mpi)
  (environment=(MPI_MAX_CLUSTER_SIZE 1))
  (directory="/home/radu/APPS/HANS")
  (executable="bw_halo_sis")
```

Example 7.13 (Configuration file - `bench.in`).

```
!ZEN$ SUBSTITUTE threads = { 1:4 }
threads
```

Three ZEN directives have been inserted into two files which specify 40 experiments automatically generated and conducted by ZENTURIO. Figure 7.6(a) displays a good scalability of this code. Backward pricing is a computational intensive application, which highly benefits from the inter-node

MPI and intra-node OpenMP parallelisation. The overall wallclock time of the application significantly improves by increasing the number of nodes and the OpenMP threads per SMP node. Figure 7.6(b) displays a very high ratio between the total execution user time (i.e., one full bar) and the HPF and the MPI overheads, which explains the good parallel behaviour. This ratio decreases for a high number of SMP nodes, for which the overheads significantly degrade the overall performance.

7.1.5 Three Dimensional FFT Benchmarks

The performance of parallel scientific applications is heavily influenced by various mathematical kernels, like linear algebra software [174] that needs to be individually optimised for each particular platform to achieve acceptable high performance In this context, ZENTURIO has been deployed at the Paul Scherrer Institute (Swiss Federal Institute of Technology – ETH Zurich) for automatic benchmarking of three-dimensional FFT kernels required by a group of physicists for solving large-scale partial differential simulations [122]. This section reports experimental results produced by this international synergy effort.

Let $A(n,n,n)$ denote a three dimensional array. A three dimensional FFT transform on the array A is defined as:

$$B_{x,y,z} = \sum_{s=0}^{n-1}\sum_{t=0}^{n-1}\sum_{u=0}^{n-1} \omega^{\pm(xs+yt+zu)} A_{s,t,u}, \ \forall\, x,y,z \in [0..n-1],$$

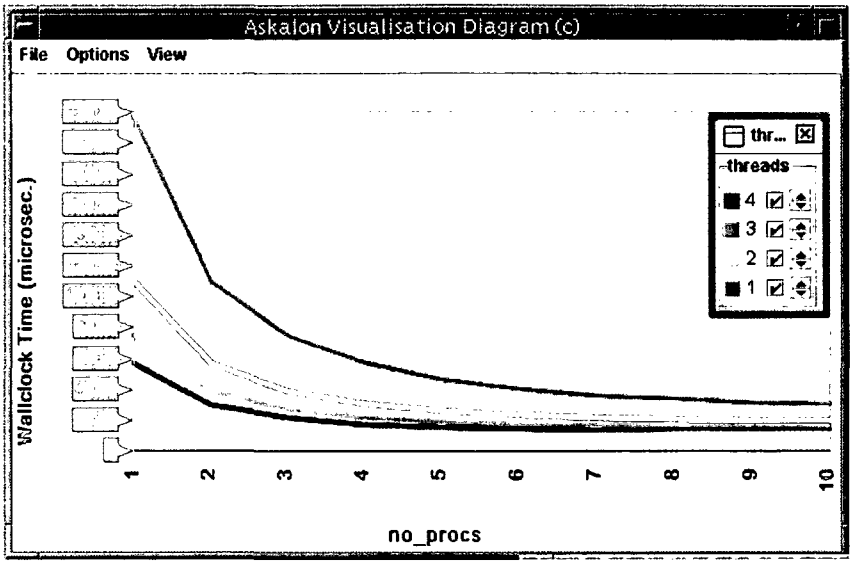
where $n = 2^m$ and $\omega = e^{\frac{2\pi i}{n}}$ is the n^{th} root of unity. The computation is parallelised by distributing the x dimension of the cube onto the array of available processors (see Figure 7.7). As a consequence, the computation over the inner y and z dimensions can be performed in locally on each processor in parallel independent loops given by the following first two equations:

$$\begin{aligned} C_{s,t,z} &= \textstyle\sum_{u=0}^{n-1} \omega^{zu} A_{s,t,u}; \\ D_{s,y,z} &= \textstyle\sum_{t=0}^{n-1} \omega^{yt} C_{s,t,z}; \\ B_{x,y,z} &= \textstyle\sum_{s=0}^{n-1} \omega^{xs} D_{s,y,z}. \end{aligned}$$

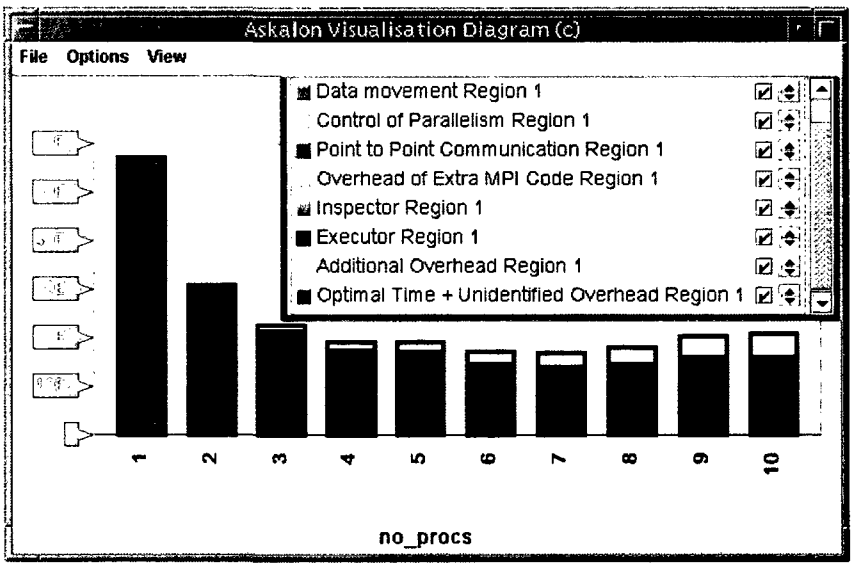
The summation on the x axis, expressed by the last equation above, requires redistribution of the matrix elements such that each processor can compute its sum locally. This is done by rotating the cube around the z dimension in an operation called *transpose*. Finally, a second reverse transpose operation is required to rearrange the data to the original layout (see Figure 7.7).

This section presents a comparative analysis between two three-dimensional FFT implementations, as follows.

1. *FFTW* [73] is a portable subroutine library for computing the Discrete Fourier Transform in one or more dimensions of arbitrary input sizes, and



(a) Wallclock time for various intra-node and inter-node machine sizes.



(b) Contribution of the MPI and the HPF overheads to the wallclock time, for various inter-node machine sizes, four threads per SMP node.

Fig. 7.6. Benders decomposition performance results for various machine sizes.

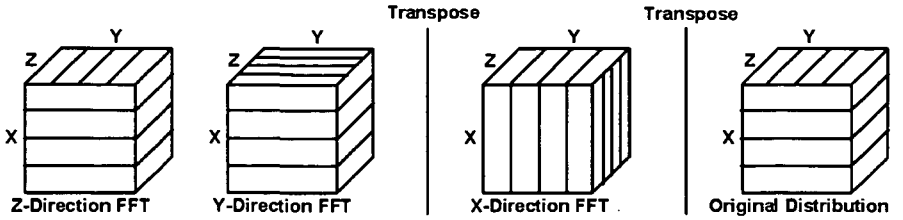


Fig. 7.7. The parallel three-dimensional FFT computation.

of both real and complex data. Existing benchmarks [72] performed on a variety of platforms show that the performance of FFTW is typically superior to that of other publicly available FFT software, and is even competitive with non-portable, highly optimised vendor-tuned codes. The power of FFTW is the ability to optimise itself to the target machine through some pre-defined codelets run by a planner function before calling the real FFT;

2. *wpp3DFFT* developed by Wes Petersen at ETH Zurich uses a generic implementation of Temperton's in-place algorithm [155] for an $n = 2^m$ problem size, with the particular focus of making the transpose faster. The optimised algorithm pays a flexibility price, which restricts the problem matrix size and the machine size to the powers of two.

Both applications are encoded as C MPI programs, which were wrapped by a Fortran front-end with the purpose of using the automatic compiler-based instrumentation provided by ZENTURIO and SCALEA. All the experiments have been conducted on a single Intel Pentium III Beowulf cluster at ETH Zurich, which comprises 192 dual CPU Pentium III nodes running at 500 MHz with 1GB RAM, interconnected through 100 MBit per second Fast Ethernet switches. The nodes are organised into 24 node frames interconnected through 1 GBit per second optical links.

The following three application parameters have been varied:

1. *The problem size* ranges from 2^3 to 2^8, which is expressed by the ZEN variable `problemsize` in Example 7.14. Larger problem sizes could not be run due to the limited amount of memory available on one cluster node.
2. *The communication library* is expressed by the `MPI_HOME` ZEN variable in the application `Makefile` (see Example 7.15). The communication libraries under comparative study are the LAM [26] and the MPICH (using the P4 communication device) [79] MPI implementations. Shared memory has been used for the communication within one SMP node.
3. *The machine size* ranges from 2^1 to 2^6 dual nodes, each node running two MPI processes. The `MPIRUN` ZEN variable refers to the implementation specific `mpirun` script which is not standardised by MPI. The constraint directive insures the correct association between the `mpirun` script and the

MPI library location, which is parameterised externally in the `Makefile`. Larger machine sizes have been limited by the cluster queuing policy.

4. *The performance metrics* of interest are the total execution time and the transpose time which were measured using the ZEN performance directive illustrated in Example 7.14. The MPI communication overheads have been measured using the SCALEA MPI wrapper library.

Since small FFT problems have extremely short execution times (i.e., order of milliseconds), they are prone to perturbations coming from the operating system or other background processes that run with low scheduling priority. To avoid such consequences, each experiment is repeated for a long enough time (i.e., five minutes) and the mean of all measurements is computed.

Example 7.14 (FFT Fortran wrapper – `FLauncher.f`).

```
problemsize=64
*ZEN$ ASSIGN problemsize = { 2**{3:8} }
minutes=5
call MPI_INIT(ierr)
call pre_measure(problemsize, minutes)
*ZEN$ CR wpp3dfft PMETRIC WTIME BEGIN
call to_measure()
*ZEN$ END CR
call post_measure()
call MPI_FINALIZE(ierr)
```

Example 7.15 (`Makefile`).

```
MPI_HOME = /usr/local/apli/lam
#ZEN$ ASSIGN MPI_HOME = { /usr/local/apli/lam,
                          /usr/local/apli/mpich }
. . .
$(EXEC): $(OBJS)
  $(MPI_HOME)/bin/mpicc -o $(EXEC) $(OBJS) $(LIBS)
```

Example 7.16 (PBS script – `run.pbs`).

```
#!/bin/sh
#ZEN$ SUBSTITUTE nodes\=1 = { nodes\={2,4,8,16,32,64} }
#PBS -l walltime=3600,nodes=1:ppn=2
nproc=`wc $PBS_NODEFILE | awk '{print $1}'`
LAM_RUN="/usr/local/apli/lam/bin/mpirun -np $nproc wpp3DFFT"
MPICH_RUN="/usr/local/apli/mpich/bin/mpirun -nolocal
          -np $nproc -machinefile $PBS_NODEFILE wpp3DFFT"
/usr/local/apli/lam/bin/lamboot -v $PBS_NODEFILE
MPIRUN=$LAM_RUN
#ZEN$ ASSIGN MPIRUN = { $LAM_RUN, $MPICH_RUN }
#ZEN$ CONSTRAINT INDEX MPIRUN == Makefile:MPIHOME
$MPIRUN
```

A total of six ZEN directives have been inserted into three application files to express 72 experiments automatically generated and conducted by ZENTURIO. Figures 7.8(a) and 7.8(b) display the speedup curves of the two FFT algorithms, normalised against the lowest machine size executed (i.e., two dual nodes), since a sequential experiment was not available. The speedup is bad for small problem sizes for which large parallelisation deteriorates the performance. Large problem sizes offer some speedup until a certain critical machine size.

The explanation for the poor speedup curves is given by the large fraction used by the transpose operation (i.e., region 2) and the MPI overheads (i.e., `MPI_Sendrecv_replace` routine used to interchange the elements in the transpose) from the overall execution time, as displayed in Figure 7.9(a) (FFTW shows similar overhead curves). It is interesting to notice that both algorithms scale quite well until 16 dual nodes for a 2^8 problem size, after which the performance significantly degrades. The reason is the fact that larger machine sizes spawn across multiple cluster frames which communicate through 3 PCI switches, 2 Ethernet, and 2 Fast-Ethernet wires that significantly affect the transpose communication time. For small problem sizes, the execution time is basically determined by the transpose overhead that naturally increases proportional with the machine size (see Figures 7.10(a) and 7.9(b)). In contrast to wpp3dFFT, FFTW shows an interesting behaviour of keeping the transpose and the total execution time constant even for large machine sizes. The explanation is given by the load balancing analysis which is explained in the next paragraph.

ZENTURIO offers a series of data aggregation functions, comprising maximum, minimum, average, or sum, for metrics measured within the parallel (MPI) processes or the parallel (OpenMP) threads of an application.

Definition 7.17. *Let $\mathcal{M}$ denote a performance metric and $\mathcal{M}_i$ its measured instantiations across all n parallel processes or threads of a parallel application, $\forall\, i \in [1..n]$. The* load balance *aggregation function for the metric $\mathcal{M}$ is defined as the ratio between the average and maximum aggregation values:*

$$\frac{\frac{\sum_{i=1}^{n} \mathcal{M}_i}{n}}{\max\limits_{\forall i \in [1..n]} \{\mathcal{M}_i\}}.$$

The wpp3dFFT kernel shows a good load balance close to one for all the problem and the machine sizes examined (see Figure 7.12(b)), while FFTW exhibits a severe load imbalance behaviour, the smaller problems are and the larger the machine sizes get (see Figure 7.12(a)). The explanation is the fact that FFTW in its planner function (that chooses optimised codelets for a certain platform) also detects that a machine size is too large for a rather small problem size to be solved. As a consequence, it decides to use only a subset of the processors for doing useful computation and transpose, while the remaining MPI processes simply exit by calling the `MPI_Finalize` routine.

This explains the even execution time for small problem sizes which was shown in Figure 7.10(a).

Figure 7.11(a) shows a better performance of the LAM MPI implementation compared to MPICH for small problems and large machine sizes. Such experiments are bound to exchanging large number of small messages dominated by latencies, for which the LAM implementation seems to perform better. Large problem sizes shift the focus from message latency to network bandwidth, in which case both implementation perform equally well (see Figure 7.11(b)).

A complementary suite of experiments performed on a different cluster using the technique already presented in Example 7.8 (see Section 7.1.2) shows that the Myrinet high performance interconnection network (not available on the ETH cluster) gives an approximate two fold improve in performance compared to the Fast Ethernet (see Figure 7.10(b)).

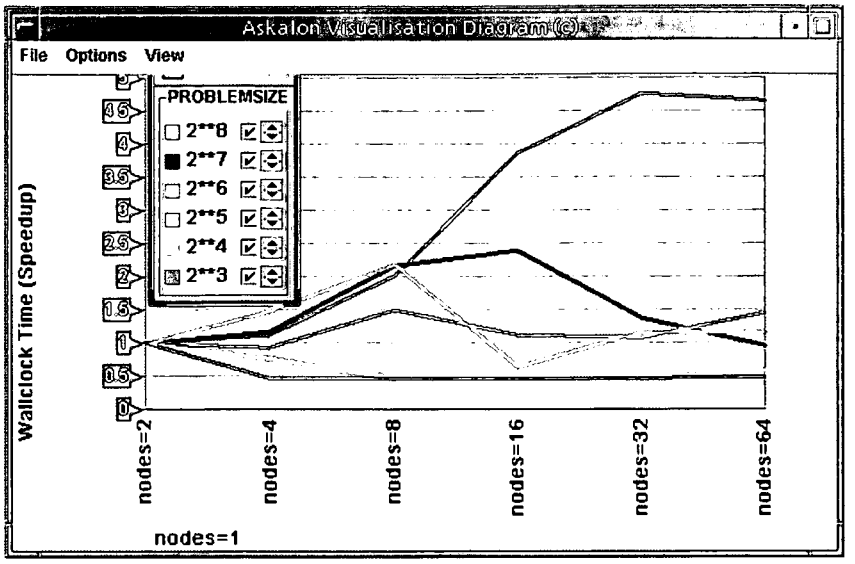
A comparative analysis of the two FFT parallel algorithms shows, as expected, a better performance of wpp3DFFT compared to FFTW for large problem sizes, which is due to the highly optimised wpp3DFFT transpose implementation (see Figure 7.13(a)). For small problem sizes, FFTW performs much better due to its intelligent run-time adjustment of machine size in the planning phase (see Figure 7.13(b)). The metric in which the ETH physicists are particularly interested is the ratio between the transpose and computation time, the latter being defined as the difference between the overall execution time and the transpose operation. This metric is comparatively displayed in Figures 7.14(a) and 7.14(b).

7.1.6 Registry Service Throughput

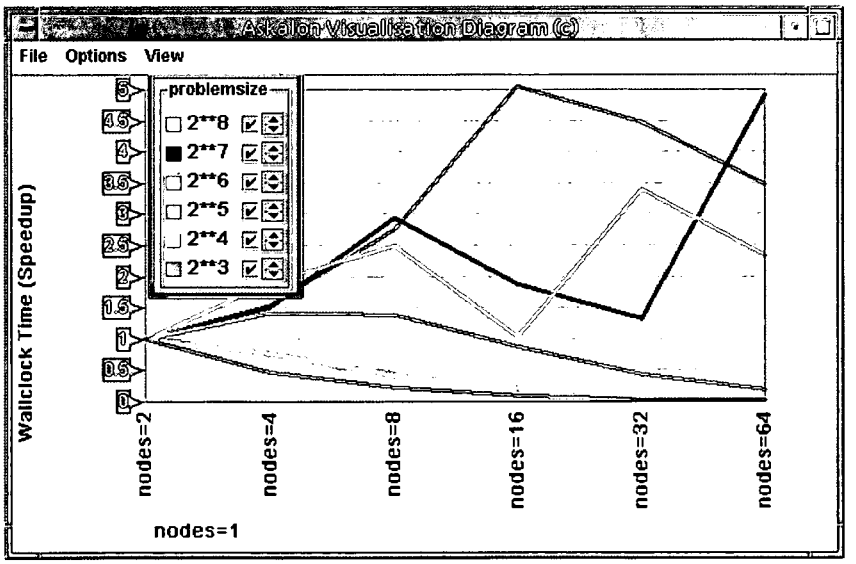
Section 5.8 has presented a comparative analysis between the WASP-based and the OGSI-based implementations of the ZENTURIO Grid services, which includes a Registry service for transient service registration and discovery. Both the WASP-based Registry (see Section 5.3.5) and the Globus VORegistry provide a flat service organisation which is subject to scalability limitations. The purpose of this section is to comparatively explore the responsiveness of these two alternative Registry service implementations under a heavy service registration and client lookup load.

The scalability benchmark has been automatically conducted using ZENTURIO by running both the client and the hosting environment on a four processor 750 MHz SMP Sun-fire with 9GB memory, to avoid CPU contention and network delays. The experiments have been specified by annotating the benchmark client application with ZEN directives as shown in Example 7.18. The following annotations have been performed:

1. *The number of registered services* from 100 to 1500 with the stride 100, denoted by the ZEN variable `svNo`;
2. *The number of concurrent clients* from 100 to 15100 with the stride 1000, denoted by the ZEN variable `clnts`;

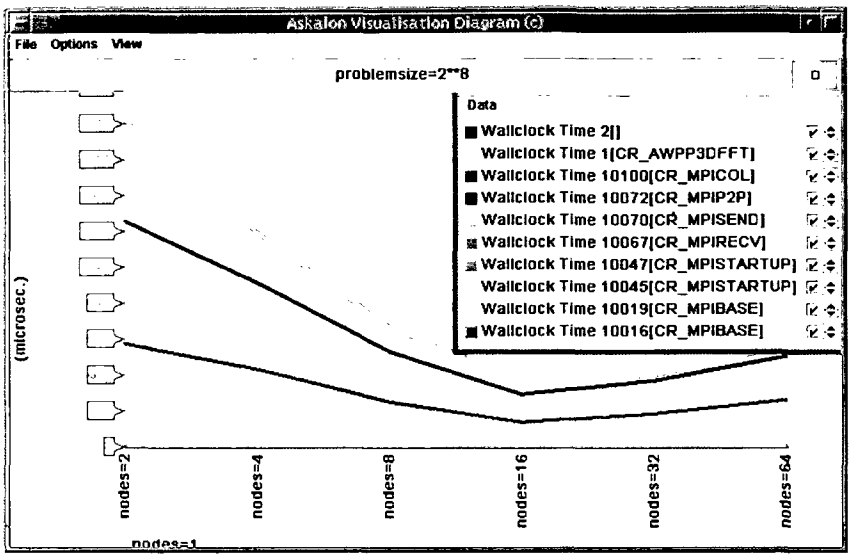


(a) FFTW speedup.

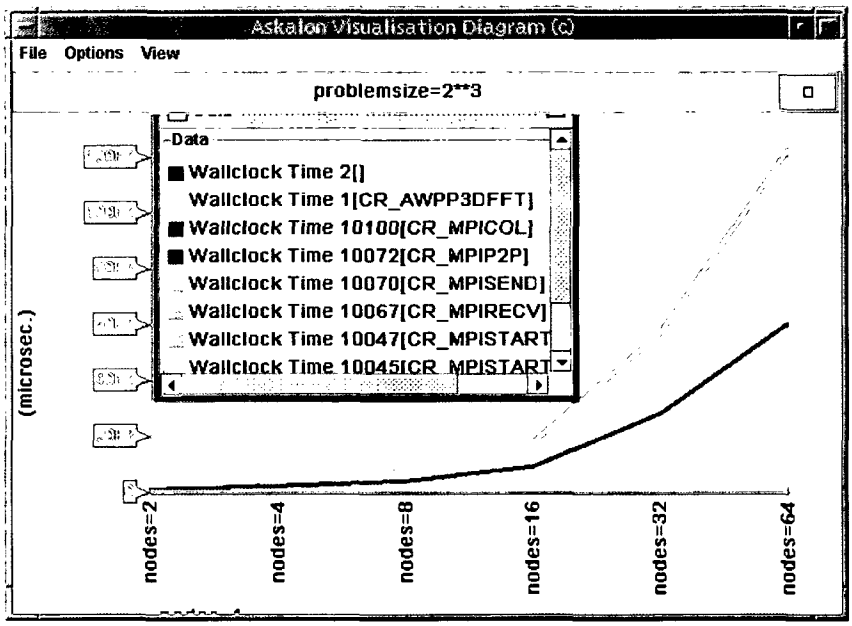


(b) wpp3DFFT speedup.

Fig. 7.8. Three-dimensional FFT benchmark results (I).

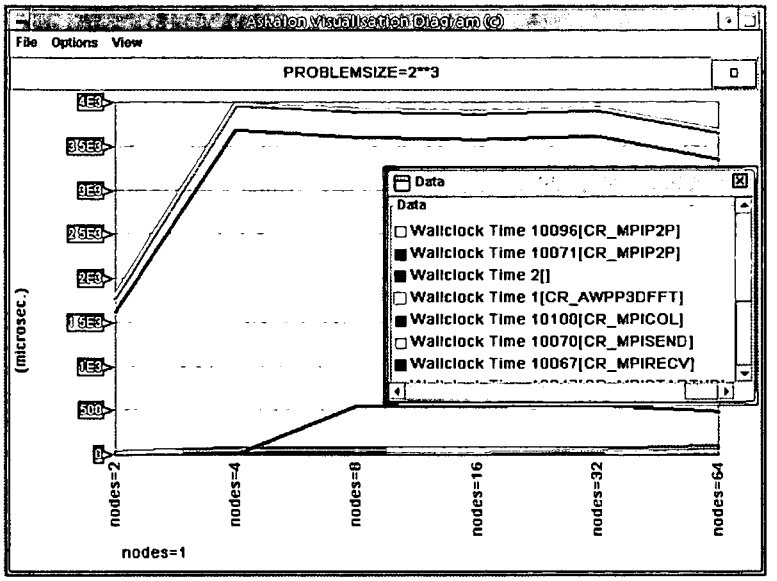


(a) wpp3DFFT overheads (2^8 problem size).

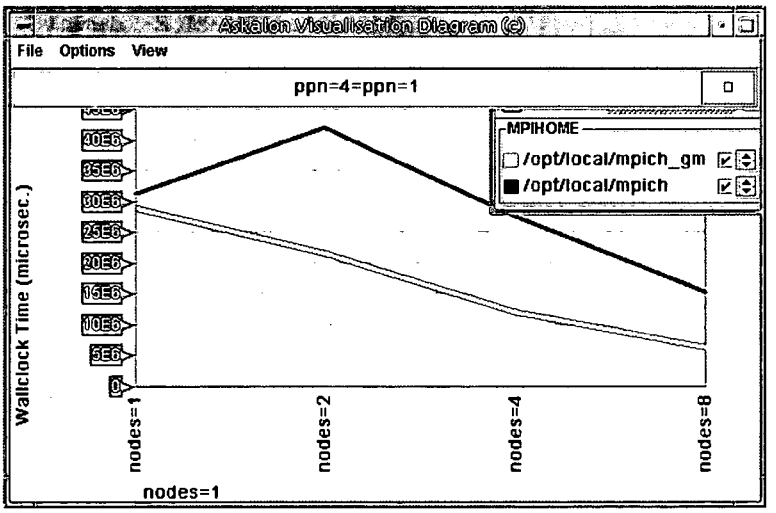


(b) wpp3DFFT overheads (2^3 problem size).

Fig. 7.9. Three-dimensional FFT benchmark results (II).

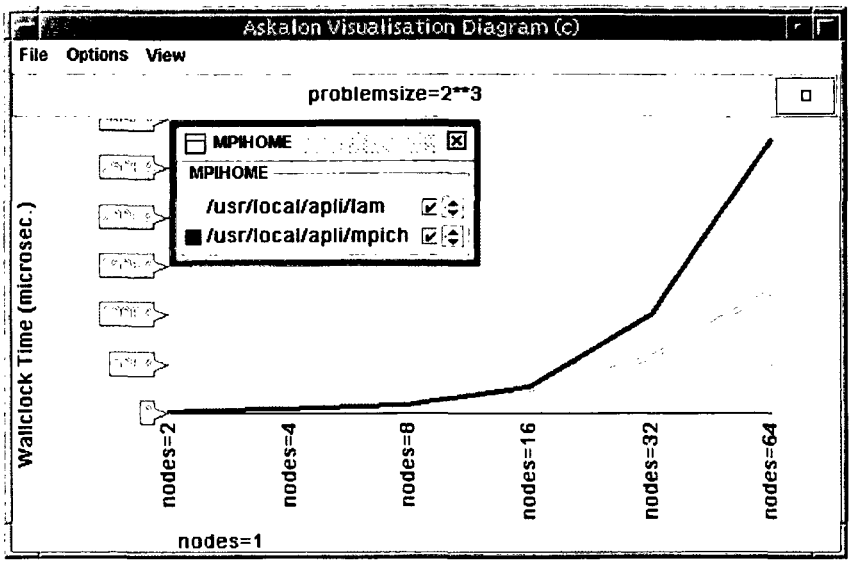


(a) FFTW overheads (2^3 problem size).

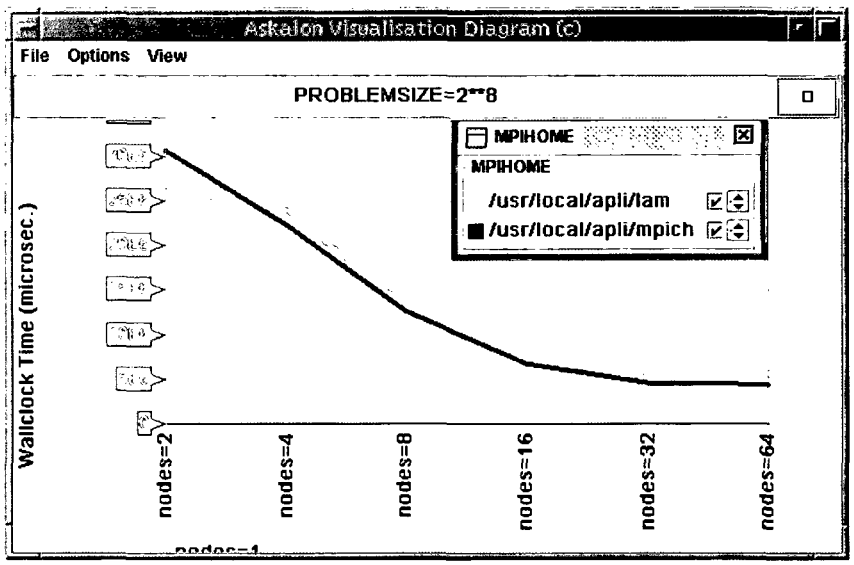


(b) wpp3DFFT network comparison (LAM versus MPICH, 2^8 problem size).

Fig. 7.10. Three-dimensional FFT benchmark results (III).

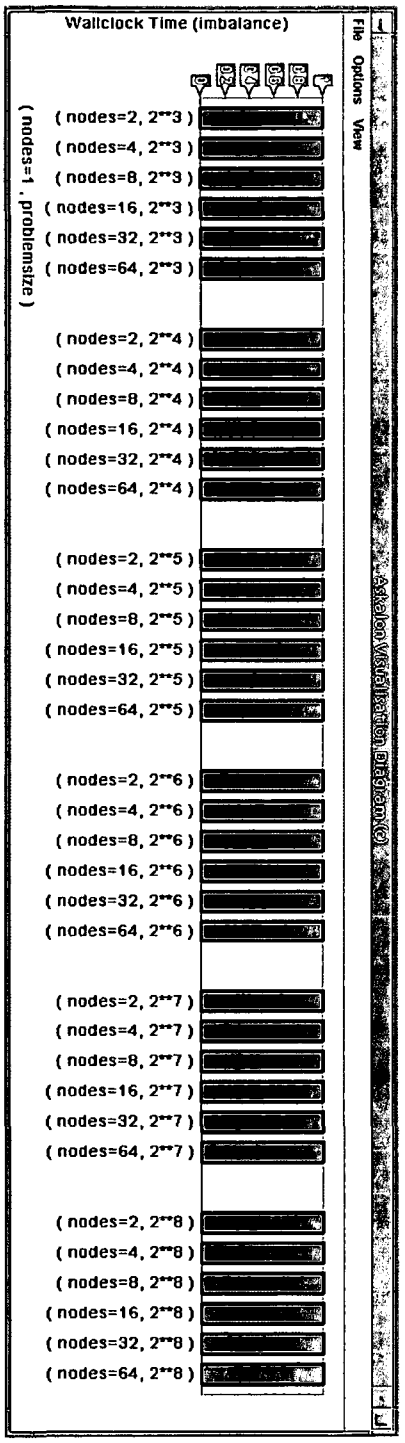


(a) wpp3DFFT network comparison (LAM versus MPICH, 2^3 problem size).

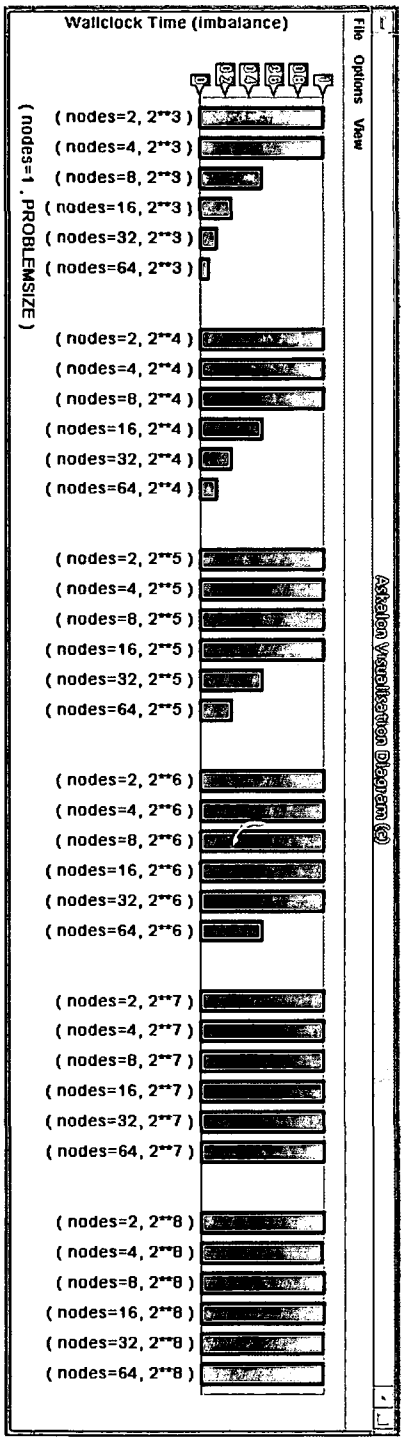


(b) FFTW network comparison (LAM versus MPICH, 2^8 problem size).

Fig. 7.11. Three-dimensional FFT benchmark results (IV).

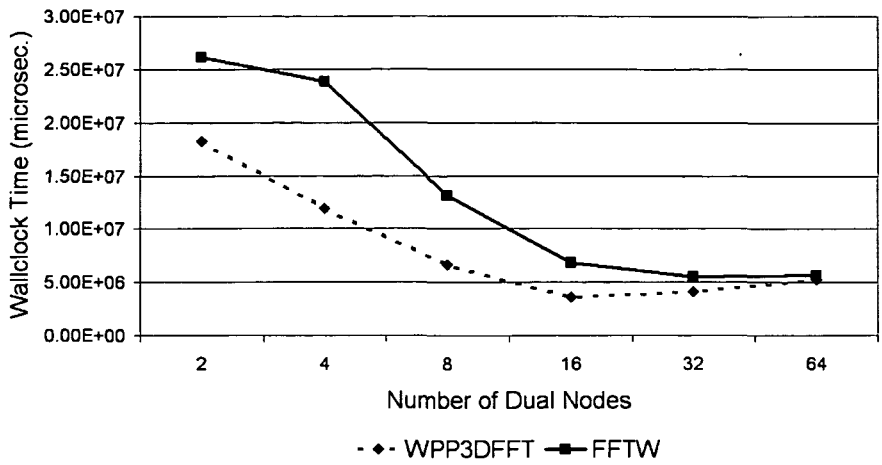


(b) wpp3DFFT load balance.

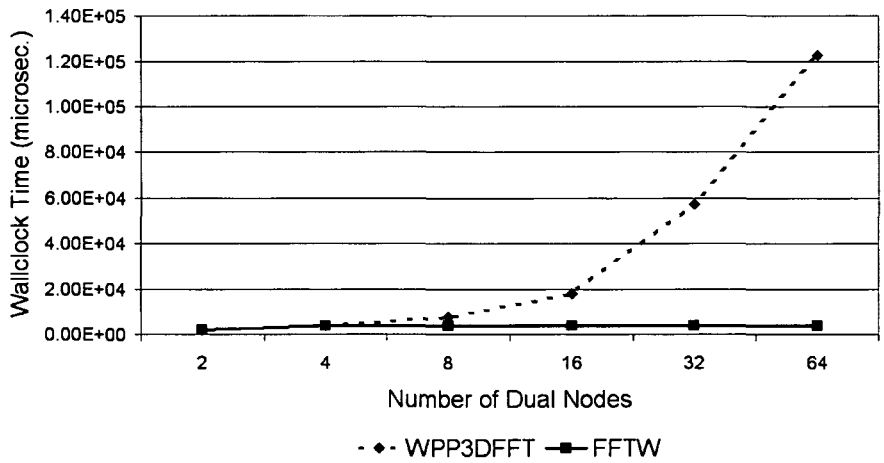


(a) FFTW load balance.

Fig. 7.12. Three-dimensional FFT benchmark results (V).

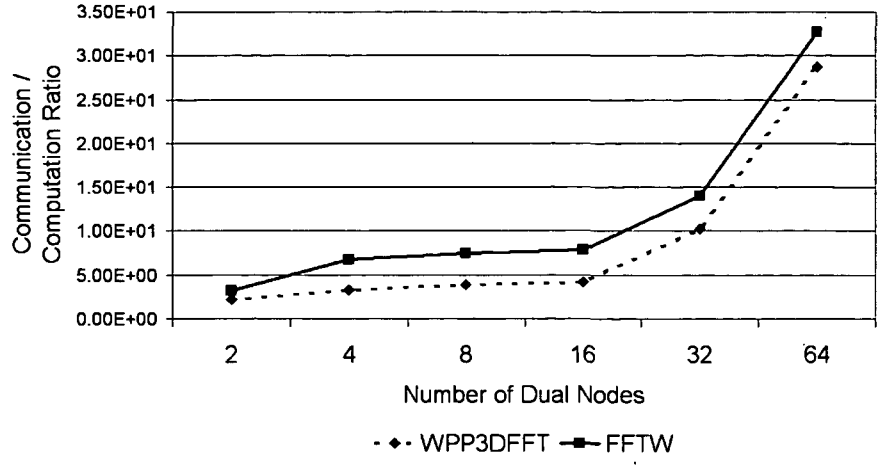


(a) wpp3DFFT versus FFTW, 2^8 problem size).

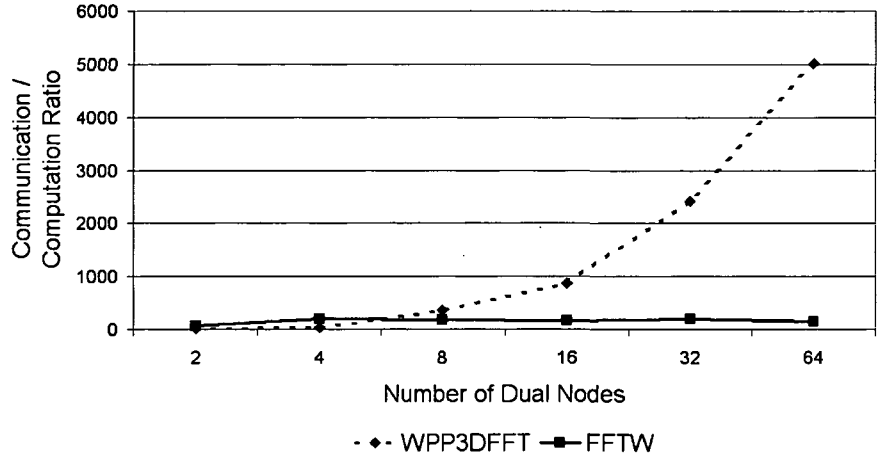


(b) wpp3DFFT versus FFTW, 2^3 problem size).

Fig. 7.13. Three-dimensional FFT benchmark results (VI).



(a) wpp3DFFT versus FFTW, 2^8 problem size.



(b) wpp3DFFT versus FFTW (2^3 problem size).

Fig. 7.14. Three-dimensional FFT benchmark results (VII).

3. *The number of requests per second* served by the Registry have been measured by manually instrumenting the client with the SCALEA instrumentation library.

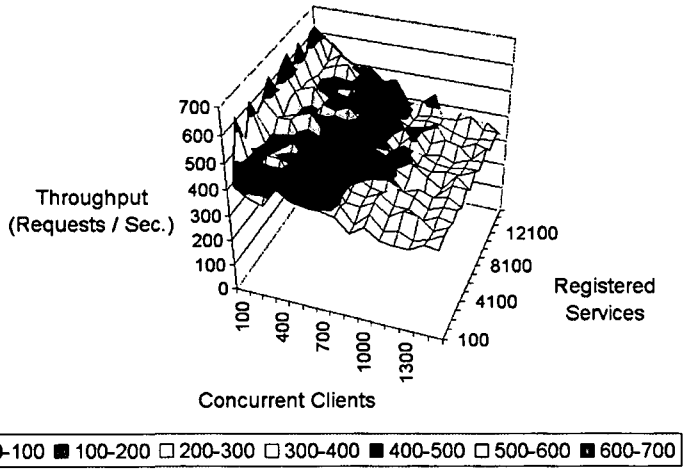
These annotations specify a total of $15 \times 15 = 225$ experiments which were automatically generated and conducted by ZENTURIO.

Example 7.18 (OGSI VORegistry benchmark client.).

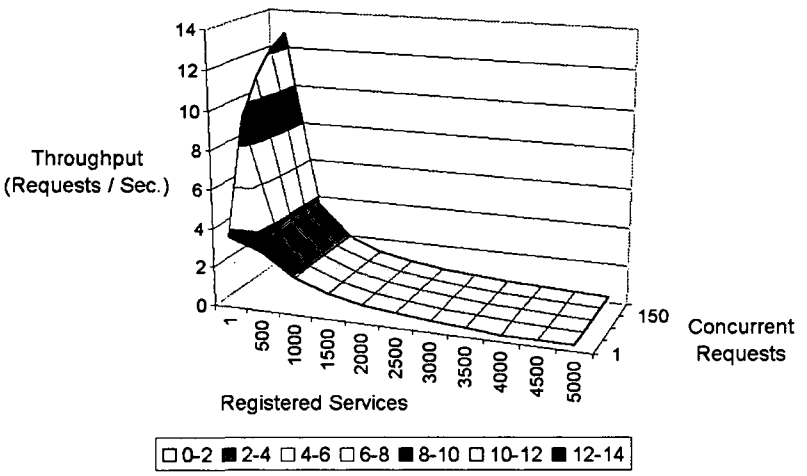
```
. . .
svNo = 100;
//ZEN$ ASSIGN svNo = { 100 : 1500 : 100 }
for(int i = 0; i <= svNo; i++) {
  ((Stub) factory)._setProperty(
                  ServiceProperties.INVOCATION_ID, i);
  factory.createService(new CreationType());
}
. . .
clnts = 100;
//ZEN$ ASSIGN clnts={ 100 : 15100 : 1000 }
for(int j = 0; j < clnts.length; j++) {
  new Thread() { public void run() {
    ExtensibilityType queryResult = registry.
      findServiceData(QueryHelper.getXPathQuery(
                    "GridServiceRegistryWSInnspection
                    XPathExpr, namespaces));
    }.start();
}
```

The WASP-based Registry offers an excellent throughput of approximately 600 requests per second for around 300 concurrent requests (see Figure 7.15(a)). As expected, the performance decreases to about 300 requests per second in the case of 1500 concurrent clients. The number of registered services does not influence the overall performance due to the hash-based service organisation. The various peaks in the graph are due to the Java management of this memory intensive application and the occasional garbage collection invocations. The expectation is to see the same sustained performance for higher number of services, until the memory limits are reached and the Registry starts swapping. Since the experiments have been conducted on the main file server of the Institute for Software Science, University of Vienna, this extreme case has been omitted on purpose.

For the OGSI VORegistry, the throughput of the service lookup operations based on `findServiceData` XPath queries rapidly decreases with the number of registered services (see Figure 7.15(b)). The reason is the sequential organisation of the service data elements into a single XML document, which is clearly not a scalable approach for high-throughput delivery.



(a) WASP-based Registry.



(b) OGSI-based VORegistry.

Fig. 7.15. Sustained Registry throughput results.

7.1.7 Grid Service Throughput

During the testing phase of the OGSI-based ZENTURIO prototype on various performance and parameter studies, a severe decrease in the responsiveness of the overall Grid services coordination workflow compared to the WASP-based version was clearly visible. The performance was particularly poor in two situations:

1. when the Experiment Generator service generates experiments at a high rate and passes them immediately to the Experiment Executor service for execution;
2. when many notification events are sent to the User Portal at the same time as a result of multiple experiments changing state simultaneously.

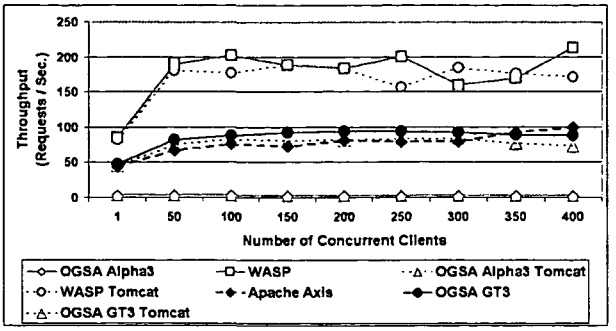
This section aims to conduct a small automatic benchmark test using ZENTURIO that compares the WASP and OGSI service throughput. The purpose is not to perform a fair benchmark between the two SOAP implementations, nor to debug their internals to detect the real cause of the performance bottleneck, rather to highlight an existing OGSI performance bug.

The service throughput in requests per second has been measured for the following three different SOAP invocations:

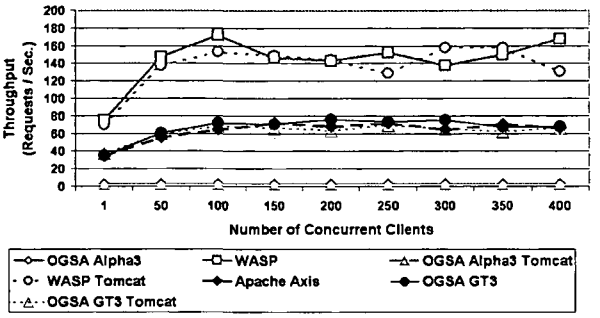
1. an array of 100 elements;
2. a string of 100 characters;
3. an array of 100 strings of 100 characters each.

There is no input argument to the requests (i.e, not an echo test), because most of the real-world web applications will send small requests most of the time. All the experiments have been performed on a four processor 750MHz SMP Sun-fire with 9GB memory to avoid network delays and CPU contention between the client and the hosting environment. The default serialisers and SOAP encodings of each (i.e., WASP and Axis) SOAP engines have been used. The array and the string structures have been pre-built on the server as static data members. There is a start-up period of 100 transactions to ignore service loading and other optimisation settings specific to each hosting environment. The default own hosting environments provided by WASP and OGSI distributions have been used. For consistency, an additional test has been conducted where both SOAP platforms are deployed within the Tomcat [106] hosting environment. The same test has been also performed for vanilla Apache Axis deployed in Tomcat. The hosting environments have been properly set-up to accommodate the full amount of concurrent requests needed. The experiments have been automatically conducted by ZENTURIO using a benchmark client similar to the one shown in Example 7.18 (i.e., ZEN variable `clnts`).

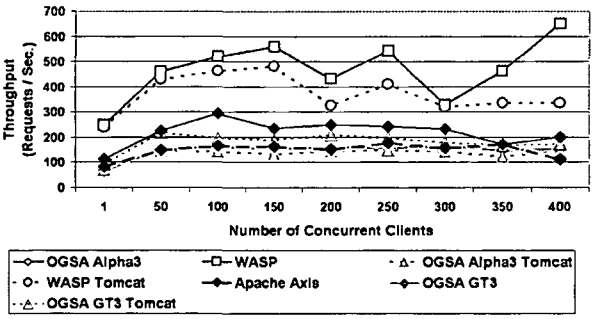
The results depicted in Figure 7.16 show that WASP is doubling the throughput offered by the Globus OGSI implementation. The object size and the memory consumption are similar in both implementations (though WASP has an overall memory usage slightly bigger). The performance differences are



(a) 100 integer array.



(b) 100 string array.



(c) 100 character string.

Fig. 7.16. Comparative sustained throughput results of WASP, OGSI, and vanilla Axis services.

due to a more mature streaming architecture offered by WASP which includes interception, XML parsing, and SOAP message processing. As expected, OGSI displays similar performance with vanilla Apache Axis, since it does not add any overhead on top of the JAX-RPC serialisation. The Tomcat deployment does not influence the results significantly, though for WASP it introduces a slight overhead. The poor performance of the OGSI alpha-3 release (which was the initial implementation platform that motivated the entire benchmark) on manipulating arrays was due to a serialisation performance problem in the underlying Axis1.1 Release Candidate 2.

7.2 Parameter Studies

Even though the original idea of ZENTURIO was to support cross-experiment performance studies of parallel applications, the general parameter specification approach taken by the ZEN language enables straight-forward classical parameter studies. A parameter study experiment has been formally defined in Section 3.9.

7.2.1 Backward Pricing

The backward pricing kernel is a parallel implementation of the backward induction algorithm which computes the price of an interest rate dependent financial product, such as a variable coupon bond. The algorithm is based on the Hull and White trinomial interest rate tree models for future developments of interest rates [46].

The application is encoded such that it reads the input parameters from different input data files. The parameter annotations for this study are performed by inserting ZEN assignment directives in the source code immediately after the input parameter `read` statements, as shown in Example 7.19. The `read` statements become therefore dead-code and will hopefully be eliminated through subsequent optimised compilation. The following four input parameters have been varied for this application:

1. *the coupon bond* denoted by the ZEN variable `coupon` (i.e., from 0.01 to 0.1 with the increment 0.001);
2. *the number of time steps* over which the price is computed, denoted by the ZEN variable `nr_steps` (i.e., from 5 to 60 with the increment 5);
3. *the coupon bond end time*, denoted by the ZEN variable `bond%end`. An additional constraint directive guarantees that the coupon bond end time is identical with the number of time steps;
4. *the length of one time step*, denoted by the ZEN variable `delta_t` (i.e., from 1/12 to 1 with the increment 1/12);
5. *the total price* is the output parameter of this application, whose variation as a function of the four input parameters is the subject of the study.

Example 7.19 (Backward pricing source file excerpt – pkernbw.f90).

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

Example 7.20 (Globus RSL Script – run.rsl).

```
+ (&
(*ZEN$ SUBSTITUTE gescher = { pc6163-c703.uibk.ac.at,
                  gescher.vcpc.univie.ac.at/jobmanager-pbs,
                  iris.gup.uni-linz.ac.at }*)
(*ZEN$ CONSTRAINT INDEX gescher == pkernbw.f90:bond\%coupon/4*)
   (resourceManagerContact="gescher")
   (count=4)
   (jobtype=mpi)
   (directory="/home/radu/APPS/Backward/V1.0")
   (executable="pkernbw")
)
```

A set of five ZEN directives have been inserted into one single source file to specify a total of 1481 experiments that are automatically generated and conducted by ZENTURIO. The experiments have been submitted onto the target execution Grid site using DUROC. To speed-up the completion of this rather large parameter study suite, the Globus RSL script has been annotated with three Grid sites where to split the throughput of this large experiment suite (see Example 7.20 and Figure 7.17):

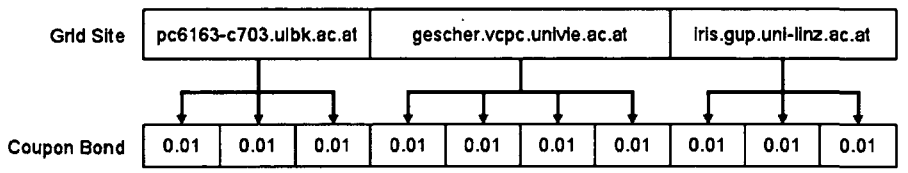


Fig. 7.17. The constraint defined in Example 7.20.

1. `pc6163-c703.uibk.ac.at` at the University of Innsbruck;
2. `gescher.vcpc.univie.ac.at` at the University of Vienna;
3. `iris.gup.uni-linz.ac.at` at the University of Linz.

The constraint directive specifies that the experiments which satisfy the condition $bond\%coupon \leq 0.03$ shall be scheduled on `pc6163-c703.uibk.ac.at`, the experiments for which $0.04 \leq bond\%coupon \leq 0.07$ shall be scheduled on `gescher.vcpc.univie.ac.at`, and the experiments having $bond\%coupon \geq 0.08$ shall be scheduled on `iris.gup.uni-linz.ac.at`. By splitting the parameter study throughput onto three Grid sites, the completion time of the whole experiment suite has been reduced by more than 50%. Section 6.4 will present a throughput scheduling approach that could replace this manual scheduling approach.

From the wide variety of visualisation diagrams automatically generated during this study, two samples are depicted in Figure 7.18. The three-dimensional surface in Figure 7.18(a) shows the evolution of the total price as a function of the number of time steps and the coupon, which can be explained as follows:

1. the price decreases with the maturity (number of time steps × length of time step), because the effect of discounting future payments increases (i.e., EU 100 in 20 years are less then EU 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);
2. the price increases with coupon, because the higher the coupon rate is, the higher the future payments are;
3. for very large maturities, the price linearly depends on the coupon only.

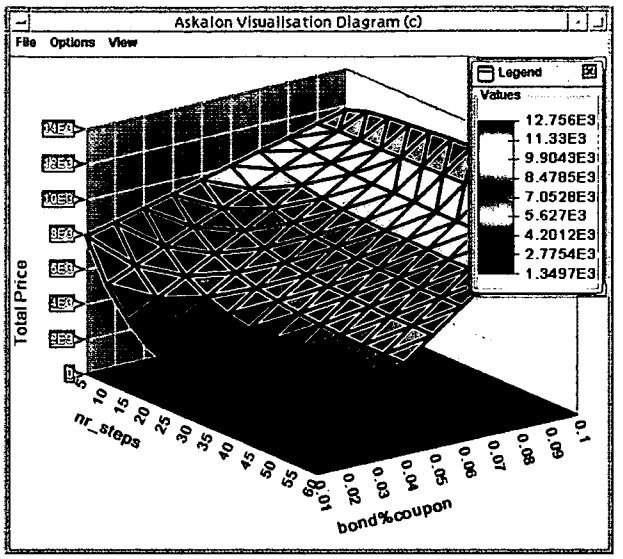
Figure 7.18(b) shows the price evolution by varying the number of time steps and the length of one time step. The interpretation of the graph is as follows:

1. the price decreases with the length of a time step, because a smaller payment number implies less money in the future;
2. depending on the number of time steps, the price may increase or decrease with the maturity, depending on how much the smaller number of payments are compensated by smaller discount effects.

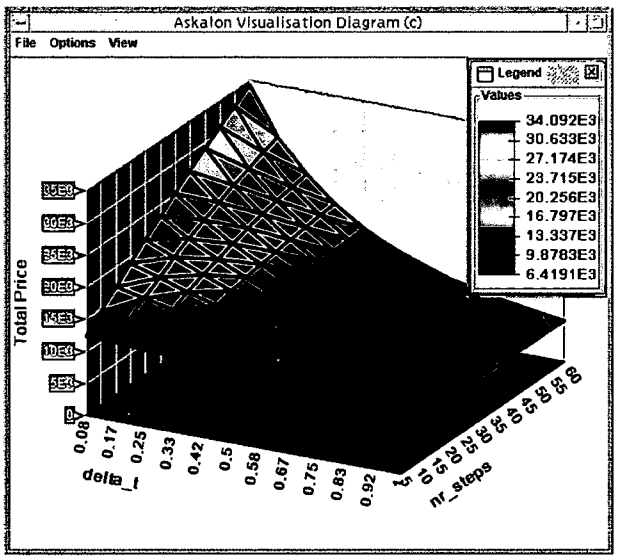
7.3 Scheduling

This section presents experimental results on two optimisation case studies discussed in Chapter 6:

1. single workflow scheduling (see Section 7.3.1);
2. throughput scheduling of independent tasks (see Section 7.3.2).



(a) Total price for `delta_t` = 1.0.



(b) Total price for `coupon` = 0.05.

Fig. 7.18. Backward pricing parameter study results.

7.3.1 Workflow Scheduling

The scheduling algorithms formally presented in Sections 6.2 and 6.3 have been applied in a Grid testbed consisting of 200 machines. To achieve a more effective evaluation of the scheduler under difficult external conditions, artificial CPU and network perturbations have been introduced (to the data delivered by NWS [176]) at random time intervals. As a consequence, the performance of the CPU and network resources of the Grid testbed follows an exponential distribution, with overloaded resources outnumbering the idle high-performance ones (which will be likely the case in future large-scale World Wide Grids).

WIEN2k

The pilot application for the workflow scheduling work is the WIEN2k [20] program package for performing electronic structure calculations of solids using density functional theory, based on the full-potential (linearised) augmented plane-wave ((L)APW) and local orbital (lo) method.

The various programs that compose the WIEN2k package are typically organised in a workflow, as illustrated in Figure 7.19. The realisation of the workflow is compliant with the model presented in Section 2.8.2. The LAPW1 and LAPW2 tasks can be solved in parallel by a fixed number of so called *k-points*, given as input parameter to the workflow orchestration program.

Reasonable accurate cost functions have been developed together with the Wien2k physicists for the most critical workflow tasks. For instance, the following analytical formulas are used to approximate the number of floating point operations of an LAPW1 and an LAPW2 k-point, respectively the file size in bytes transferred between the LAPW1 and the LAPW2 k-point computations (i.e., `case.vector`):

$$W_{LAPW1} = 7 \cdot A \cdot N^2 + N^3;$$
$$W_{LAPW2} = 10\% \cdot W_{LAPW1};$$
$$W_{12} = 200 \cdot N \cdot A,$$

where A represents the number of atoms, N represents the matrix size, and 7, 10, and 200 are scaling factors. The scaling factors of the cost functions have been adjusted to match the real execution times by conducting exhaustive multi-experimental performance studies with ZENTURIO on each workflow task using the methodology presented in Chapter 4 and exemplified on several real-world applications in Section 7.1.

Example 7.21 illustrates a sample Java CoG task graph program [7] that implements a fragment of the WIEN2k workflow application. The *JS* workflow tasks `lapw0`, `lapw1_1`, and `lapw1_2` run on the abstract machines (genetic algorithm genes) `lapw0_host`, `lapw1_host1`, respectively `lapw1_host2`. The *FT* tasks `k1` and `k2` transfer the output files of LAPW0 from `lapw0_host`

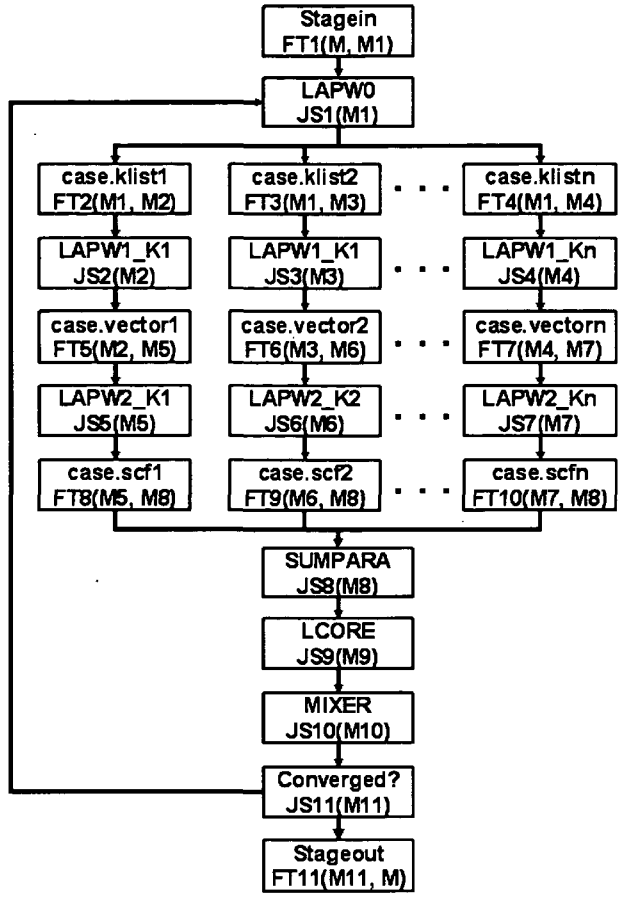


Fig. 7.19. The WIEN2k workflow.

to the abstract machines `lapw1_host1` and `lapw1_host2` where the LAPW1 k-points execute through static schedule dependencies (see Section 6.2.2).

Static Workflow Scheduling

The ZEN directives illustrated in Example 7.21 specify the possible instantiation values (representing the concrete Grid machines) of each abstract machine of the workflow. The parameter space defined by the ZEN directives is the scope of the static scheduling search algorithm. One encoding of the search engine based on genetic algorithms has been presented in Section 6.1. The manual annotation of the workflow application with ZEN directives as illustrated in Example 7.21 is, however, impractical and of little use in large-scale dynamic Grid environments. Instead, the static scheduler annotates the workflow program with ZEN directives using a special ZEN instrumentation API

provided by the Experiment Generator service (see Section 4.2). The value set of the concrete Grid machines that instantiates each ZEN variable or abstract machine is the entire set of Grid machines obtained from the Globus MDS [61]. Assuming a workflow that defines N abstract Grid machines and a Globus MDS installation that returns a set of M concrete machine, the static scheduling search space contains N^M points.

Example 7.21 (Wien2k workflow excerpt).

```
//ZEN$ SUBSTITUTE lapw0_host = { machine{1:200} }
//ZEN$ SUBSTITUTE lapw1_host1 = { machine{1:200} }
//ZEN$ SUBSTITUTE lapw1_host2 = { machine{1:200} }
. . .
Task lapw0 = createJS("lapw0_host", "lapw0");
Task lapw1_1 = createJS("lapw1_host1", "lapw1 2");
Task lapw1_2 = createJS("lapw1_host2", "lapw1 1");
Task k1 = createFT("k1","lapw0_host","lapw1_host1");
Task k2 = createFT("k2","lapw0_host","lapw1_host2");
. . .
TaskGraph taskGraph = new TaskGraphImpl();
taskGraph.add(lapw0);
taskGraph.add(lapw1_1);
taskGraph.add(lapw1_2);
taskGraph.add(k1);
taskGraph.add(k2);
. . .
Dependency dependency = new DependencyImpl();
dependency.add(lapw0.getId(), k1.getId());
dependency.add(lapw0.getId(), k2.getId());
dependency.add(k1.getId(), lapw1_1.getId());
dependency.add(k2.getId(), lapw1_2.getId());
. . .
taskGraph.setDependency(dependency);
```

Figure 7.20 depicts the generational evolution of the population best individual (i.e., static workflow schedule) for several instantiations of the static scheduling genetic algorithm applied on various WIEN2k problem size configurations. Even though the algorithm exhibits a steady smooth improvement across generations (i.e., convergence to local minima through crossover, and steep escapes from local minima through mutation), the quality of the resulting solutions is heavily influenced by several input parameters:

1. the population size;
2. the crossover probability;
3. the mutation probability;
4. the maximum generation number;
5. the steady state generation number;

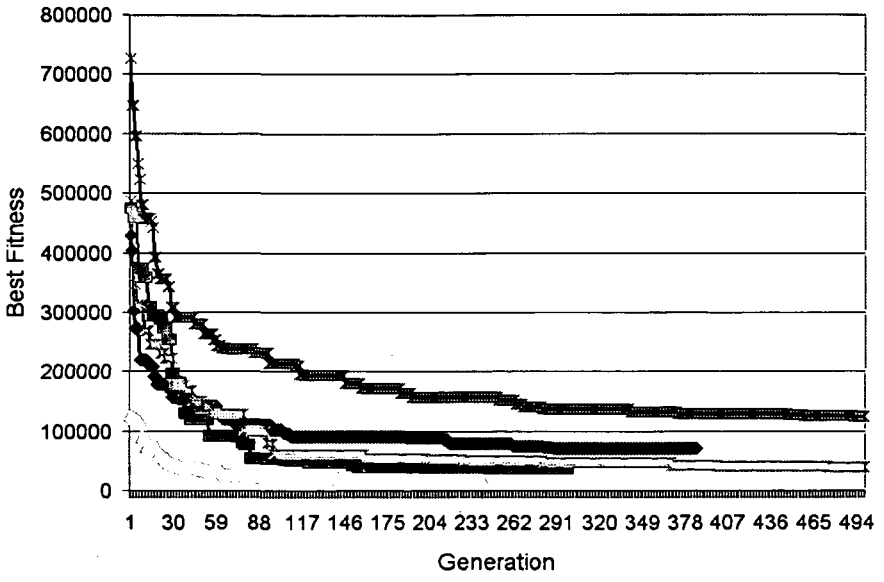


Fig. 7.20. Best individual evolution for various genetic static scheduler instances.

6. the fitness scaling factor;
7. the use of the elitist model.

A correct tuning of these parameters is crucial for the algorithm to quickly convergence to high quality solutions. In a conventional approach, this requires extensive manual experimental testing.

The genetic algorithm parameters have been tuned by conducting an aggressive exhaustive performance study using ZENTURIO in cluster mode. Seven ZEN directives that specify total of 2880 experiments have been inserted in the PBS script used by ZENTURIO to automatically generate and submit the experiments on the Beowulf cluster (see Example 7.22).

An average-sized WIEN2k workflow of about 55 nodes (i.e., 10 parallel k-points) has been used for this experiment. Every experiment represents an instance of the static scheduling algorithm configured using a different genetic parameter combination. Each static scheduling experiment annotates the application with ZEN directives that define the possible instantiations of each abstract machine, as already explained in Example 7.21. All the experiments use the Grid resource information collected at the same time instance (i.e., Grid snapshot). This hierarchical experimental setup that applies the ZENTURIO (exhaustive) performance-study tool on the ZENTURIO optimisation search engine (instantiated for the static scheduling problem) is depicted in Figure 7.21.

Example 7.22 (Genetic algorithm parameter tuning - PBS script).

```
#!/bin/sh
#PBS -l walltime=00:10:00:nodes=1
#PBS -N scheduler
size = 150
#ZEN$ ASSIGN size = { 50 : 200 : 50 }
crossover = 0.9
#ZEN$ ASSIGN crossover = { 0.4 : 1 : 0.2 }
mutation = 0.001
#ZEN$ ASSIGN mutation = { 0.001, 0.01, 0.1 }
generations = 500
#ZEN$ ASSIGN generations = { 100 : 500 : 100 }
convergence = 0.2
#ZEN$ ASSIGN convergence = { 0.1, 0.2 }
scaling = 2
#ZEN$ ASSIGN scaling = { 1, 1.5, 2 }
elitist = T
#ZEN$ ASSIGN elitist = { T, F }
${JAVA} -DSIZE=${size} -DCROSSOVER=${crossover} ...
```

The objective (fitness) function has been instantiated with the predicted workflow execution time (makespan). The workflow makespan raises the maximum optimisation difficulty since it considèrs all workflow nodes in the evaluation (e.g., for optimising communication even better results were obtained since only the *FT* tasks had to be considered). For the purpose of evaluating the quality of the solutions produced by the algorithm, the workflow makespan has been pre-measured offline on a set of idle (unperturbated) high-performance Grid resources. This will be referred in the following as *optimal fitness* $\mathcal{F}_o$. Three metrics that characterise the performance of the genetic algorithm are computed for each experiment:

1. *precision* P of the best individual $\mathcal{F}_b$ compared to the artificial optimum $\mathcal{F}_o$, defined as:

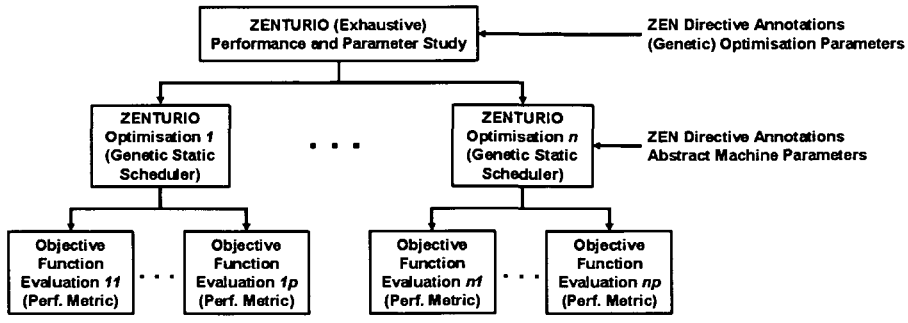


Fig. 7.21. Experimental setup for genetic static scheduler tuning.

$$P = \frac{\mathcal{F}_b - \mathcal{F}_o}{\mathcal{F}_o} \cdot 100;$$

2. *visited points* representing the total set of individuals (i.e., schedules) which have been evaluated by the algorithm during the search process;
3. *improvement* I in the fitness $\mathcal{F}_b$ of the last generation best schedule, compared to the first generation best schedule $\mathcal{F}_f$:

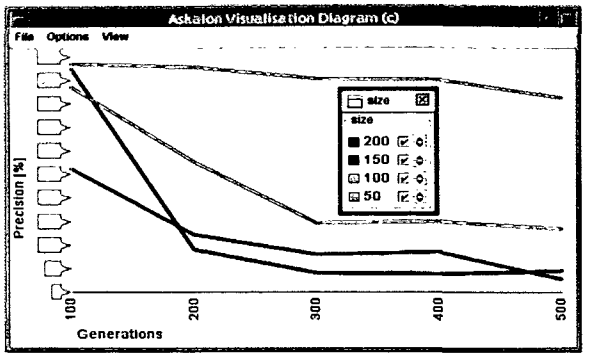
$$I = \frac{\mathcal{F}_f - \mathcal{F}_b}{\mathcal{F}_b} \cdot 100.$$

To attenuate the stochastic errors to which randomised algorithms are bound, each scheduling experiment is repeated for 30 times and the arithmetic mean of the results in each run is reported.

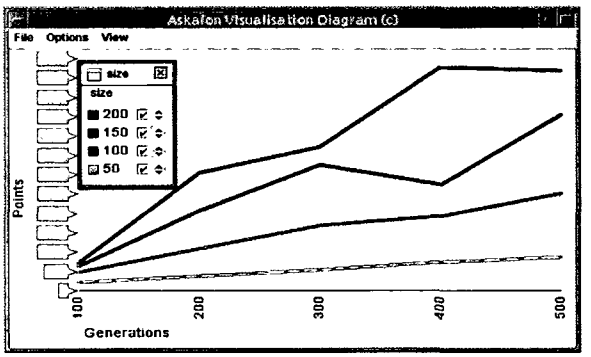
Due to the large search space (i.e., 10^{25} points) and difficult external Grid conditions (i.e., exponential resource load distribution), large populations above 50 individuals are required for converging to good solutions (see Figure 7.22(a)). As expected, the precision improves with the number of generations. Lower population sizes (e.g., 50) do not ensure enough variety in the genes and converge pre-maturely. Larger populations (e.g., 200) converge to good solutions in fewer generations, however, the number of visited points may be unnecessarily large which increases the algorithm duration. The number of visited points (i.e., the schedules computed) required for converging to good solutions is of the order of 10^4, which represents a fraction from the overall search space of 10^{25} points (see Figure 7.22(b)). The improvement in the best individual is remarkable of up to 700% over 500 generations for large populations (see Figure 7.22(c)). A value of 20% from the maximum generation number is a good effective estimate for checking whether the algorithm reached a steady state (see Figure 7.23(a)). The higher the crossover probability, the faster the algorithm converges to local maxima (see Figure 7.23(b)). A correct low mutation probability is crucial for escaping from local maxima and for obtaining good solutions (see Figure 7.23(c)). In this experiment the mutation probability had to be surprisingly low (i.e., 0.001%) due to the rather large population sizes and genes per individual (i.e., 45). Higher mutation probabilities produce too much instability in the population and chaotic jumps in the search space, that do not allow the algorithm to converge to local maxima through crossover. Fitness scaling is crucial smooth for steady improvement over large number of generations (see Figure 7.24(a)) and produces about 10 fold improvement in solution. The use of the elitist model (see Figure 7.24(b)) is beneficial due to the high heterogeneity of the search space and delivers in average 33% better solutions.

As a consequence of this performance tuning experiment, the following parameter configuration is used for the genetic algorithm within the current Grid testbed:

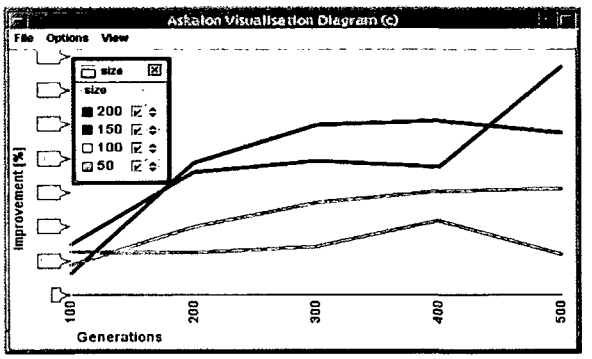
1. population size: 150;
2. crossover probability: 0.9;



(a) Population Size.

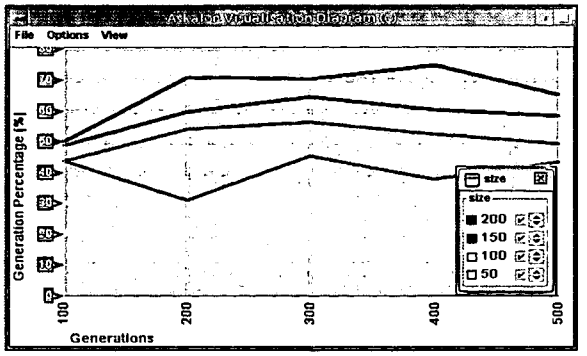


(b) Visited Points.

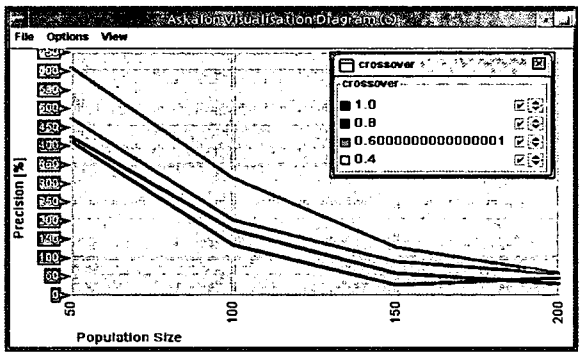


(c) Best Individual Improvement.

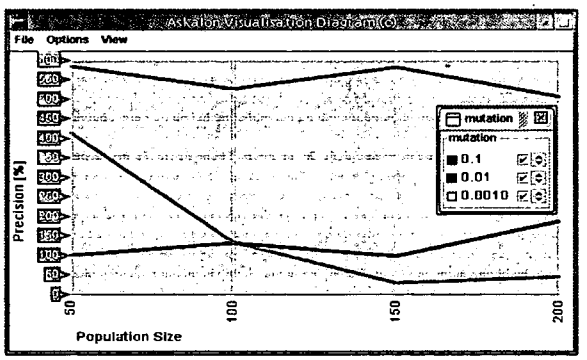
Fig. 7.22. Genetic static scheduler tuning results (I).



(a) Generation Percentage.

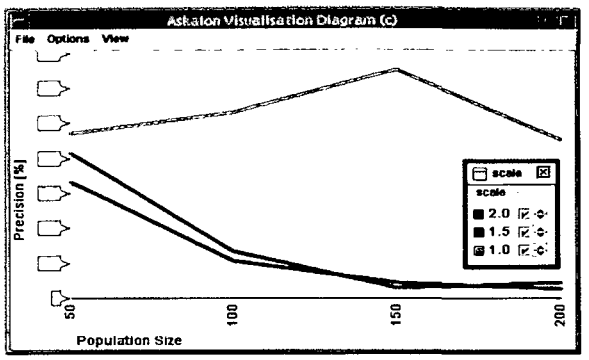


(b) Crossover Probability.

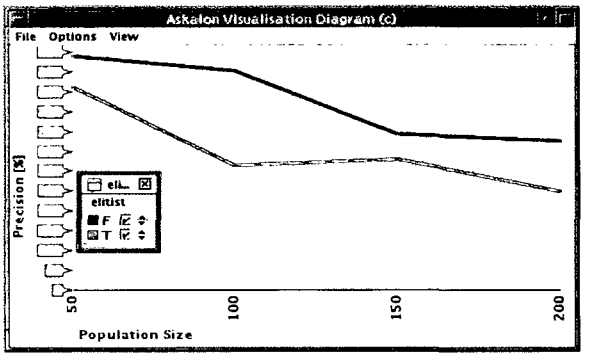


(c) Mutation Probability.

Fig. 7.23. Genetic static scheduler tuning results (II).



(a) Fitness Scaling Factor.



(b) Elitist Model.

Fig. 7.24. Genetic static scheduler tuning results.

3. mutation probability: 0.001;
4. maximum generation: 500;
5. steady state generation percentage: 20%;
6. fitness scaling factor: 2;
7. elitist model: yes.

In this configuration, the algorithm constantly delivers 25% precision and a remarkable 700% improvement in solution, by visiting a fraction (i.e., $5 \cdot 10^4$) of the entire search space points. The most sensitive parameter that needs be tuned to the workflow characteristics is the mutation probability (i.e., inversely proportional with the population size times the workflow size). The other parameter values have to be tuned to the Grid resource characteristics and are less dependent on the particular workflow.

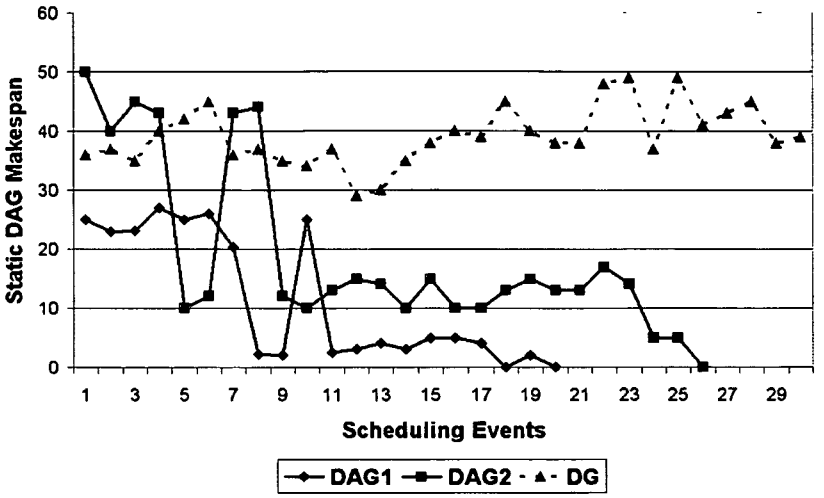
Dynamic Workflow Scheduling

The hybrid dynamic scheduling algorithm is based on the repeated invocation of the static scheduler at well defined scheduling events, in attempt to adjust the highly optimised static schedule to the dynamically changing Grid resources. In this experiment, the scheduling events are generated at the same frequency at which the artificial perturbations are introduced to the Grid computational resources (i.e., a sequential loop busy waiting a random time interval).

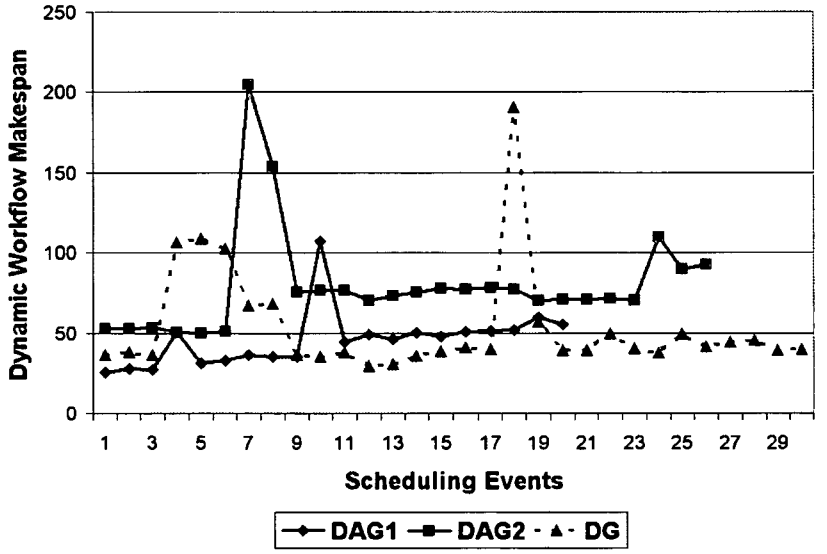
To evaluate the dynamic scheduler, three experimental WIEN2k workflow instantiations (i.e., two DAG and one DG-based) that correspond to different application input cases (i.e., the number of atoms and matrix size) with different parallelisation sizes (i.e., number of k-points) have been used. A static value of 50% is used as the performance contract elapse factor of all the workflow tasks (see Section 6.3.1).

Figure 7.25(a) traces the static DAG makespan delivered by the genetic static scheduler at consecutive scheduling events during the execution of each experimental workflow. As the workflow tasks are scheduled, execute, and terminate, the predicted static schedule makespan of the remaining DAG1 and DAG2 subworkflows obviously decreases with the number of scheduling events. The abrupt decreases of the static makespan happen after the submission of all the LAPW1 k-points (the most time consuming workflow tasks) which no longer need to be considered by the static scheduler. The abrupt increases of the makespan are due the LAPW1 tasks that violate their performance contract which need to be reconsidered by the static scheduler for rescheduling, migration, and restart. In the case of the DG-based workflow, the static scheduler always receives the complete workflow as input, but with different topological order of the nodes. This is the reason why the static makespan does not decrease with the scheduling events.

Figure 7.25(b) traces the overall predicted dynamic workflow makespan at consecutive scheduling events during the workflow execution. There are several high peaks in the histogram which are due to severe high perturbations applied to the machines running the LAPW1 k-points. As a consequence of the performance contract violation, the scheduler migrated the critical tasks to new machines at the next scheduling event, which drops the next predicted makespan close to the original predicted value. Through migration, an estimate improvement of about two fold in the overall makespan is achieved (see Figure 7.26). Since the workflow referred as DAG2 represents a larger problem size than DAG1, the benefit obtained through rescheduling and task migration is higher. The final makespan of the DAG-based workflows is, however, about twice as large as it was originally predicted by the static scheduler. While most of the performance loss is the consequence of the task restarts (i.e., due to duplicated file transfers and LAPW1 task computations), a fraction (i.e., about 10%) is due to genetic algorithm execution overhead. For the DG-based workflow, the makespan of the entire workflow could not be estimated (i.e.,



(a) Static DAG makespan.



(b) Dynamic workflow makespan.

Fig. 7.25. Dynamic scheduler workflow executions traces.

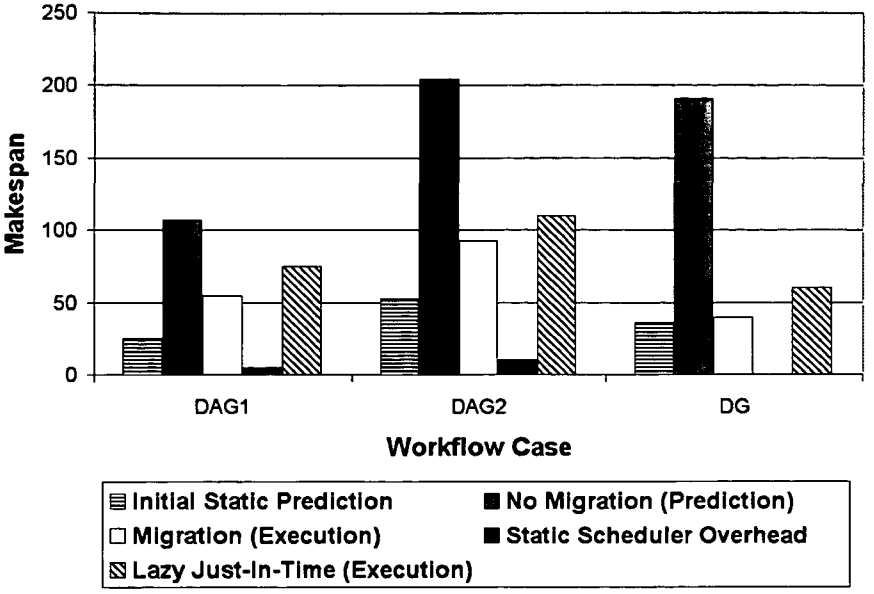


Fig. 7.26. Schedule comparison.

beyond the execution of one cycle), since the number of loops is statically unknown. As a consequence, Figure 7.25 represents the DG makespan of one workflow iteration only, which is successfully kept relatively constant through task migration in two critical occasions.

Figure 7.26 compares the hybrid approach proposed against a lazy just-in-time version of the dynamic scheduling algorithm that bypasses the static scheduler [44]. The experiments have been performed on the same workflow cases and under similar (logged) Grid conditions. Additional scheduling events are generated upon the completion of each workflow task. At each task completion scheduling event, the dependent tasks are scheduled on the resources that produce the lowest execution times ($\mathcal{O}(n)$ complexity). The overall workflow makespans obtained were in average 25% higher compared to the hybrid approach. The reason is the fact that the (genetic) static scheduler was able to find better workflow mappings by looking ahead at the entire workflow, as opposed to the lazy just-in-time scheduling of individual tasks.

7.3.2 Throughput Scheduling

The genetic algorithm for static throughput scheduling of independent tasks presented in Section 6.4 has been tested within a small simulator of Grid brokerage and task set generation. Generally, such simulators offer a more flexible and challenging testbed for appropriate algorithm validation. The *uniform*,

the *normal*, and the *exponential* distributions have been used to generate the work (i.e., floating point operations) of the task set, as well as the speed (i.e., floating point operations per second) of the resource set. The number of independent tasks used is of the order of 10^4 and the Grid size is of the order of 10^3. This produces a huge search space of 10^{30000} points and a complexity of $\mathcal{O}(10^{11})$ for the classical algorithms from the Max-min family [112] which become impractical.

The automatic creation of this large task set in Java is summarised in Example 7.23. The rather large number of ZEN directives (i.e., 10^4) is automatically inserted into the Abstract Syntax Tree of the parsed ZEN file through a special interface provided by the Experiment Generator, as already introduced in Section 4.2.

Example 7.23 (Java task set generation).

```
TaskGraph taskGraph = new TaskGraphImpl();
for(int i = 0; i < 10000; i++) {
  Task task = createTask("task" + i, "host" + i, flops());
  taskGraph.add(task);
}
taskGraph.setDependency(new DependencyImpl());
```

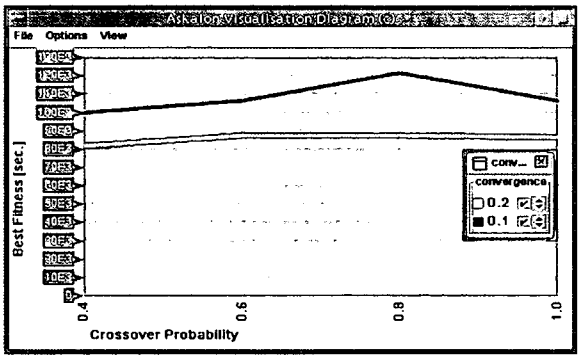
Similar to the genetic algorithm tuning for workflow static scheduling presented in Section 7.3.1, the effectiveness of the algorithm for throughput scheduling is explored for the following tunable input parameters:

1. population size;
2. crossover probability;
3. mutation probability;
4. maximum generation number;
5. steady state generation number;
6. fitness scaling factor.

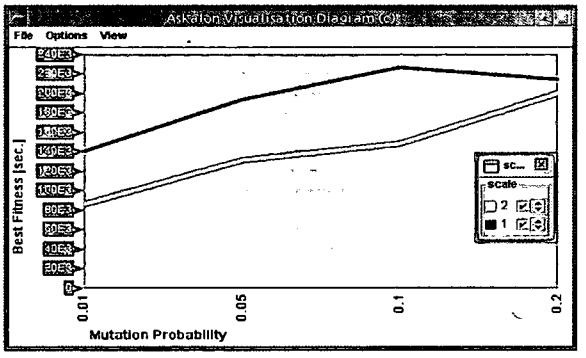
The interesting parameter values have been specified by means of ZEN directives that annotate the PBS script used to submit the experiments on a Beowulf cluster, as already shown in Example 7.22. These annotations specify a set of 1440 experiments which has been automatically generated and conducted by ZENTURIO. In order to attenuate the stochastic errors to which the randomised algorithms are bound, each experiment is repeated for 20 times and the arithmetic mean of the results in each run is reported.

An average population size of 200 individuals seems to be appropriate for a good convergence of the algorithm. A lower population size does not insure a large enough selection of individuals and causes pre-mature convergence, while larger populations slow down the algorithm with no significant improvement across generations. As expected, the quality of solution increases with the number of generations. The crossover probability does not impact the overall performance (see Figure 7.27(a)), however, a higher value insures

fast convergence to local minima. A value of 20% from the maximum number of generations seems to be an appropriate window size for checking whether the algorithm reached a steady state, after which no more improvements are being made. A low value of mutation is very effective in taking the algorithm out of local minima (see Figure 7.27(b)). A high mutation probability causes too much instability in the population due to chaotic evolution of individuals that no longer manage to steadily improve to a global maximum. Fitness scaling causes about four fold improvement in the quality of solution and must always be used.



(a) Crossover versus steady state generation percentage.



(b) Mutation versus scaling factor.

Fig. 7.27. Sample genetic algorithm tuning diagrams.

As a consequence of the tuning process, the following genetic algorithm configuration parameters are used in the proposed simulated Grid testbed:

1. population size: 200;
2. crossover probability: 0.9;
3. mutation probability: 0.0001;
4. maximum generation: 500;
5. steady state generation percentage: 20%;
6. fitness scaling factor: 2.

The results of running the algorithm on 10^4 tasks and 10^3 Grid machines, with normal, uniform, and exponential distributions of task floating point operations (i.e., between 10^3 and 10^6 flops) and machine performance rate (i.e., between 10^9 and 10^{11} flops per second), are summarised in Table 7.1. The algorithm can produce an up to 5 fold improvement of the best individual over 500 generations, by visiting less than 25000 from the overall 10^{30000} points.

Tasks	Grid	Generation	Points	Improvement	Percentage
uniform	normal	500	25000	240,92	100%
uniform	uniform	446	22300	509,90	89,2%
uniform	exponential	500	25000	222,40	100%
normal	normal	357	17850	353,12	71,4%
normal	uniform	385	19250	434,75	77%
normal	exponential	500	25000	185,48	100%
exponential	normal	500	25000	164,29	100%
exponential	uniform	402	20100	437,50	80,4%
exponential	exponential	368	18400	197,90	73,6%

Table 7.1. Genetic search algorithm results for 10^4 tasks and 10^3 Grid size for various task and Grid heterogeneity distribution testbeds.

Compared to the $\mathcal{O}(10^{11})$ complexity of the classical Max-min heuristic, the genetic algorithm computes $\mathcal{O}(25 \cdot 10^7)$ task schedules and guarantees real time response. However, unless a user-defined convergence criterion (e.g., completion deadline) is provided, the algorithm does not guarantee any quality of solution. This could be further improved by combining the genetic algorithm with task resource constraints like the Condor gangmatching [133].

Figure 7.28 shows the evolution of the makespan across the generations for all the Grid and the task configurations studied. One can recognise on the diagram two interesting behavioural patterns of the genetic algorithm:

1. smooth decreases of the makespan to local minima due to crossover;
2. big drops out of the local minima due to mutation.

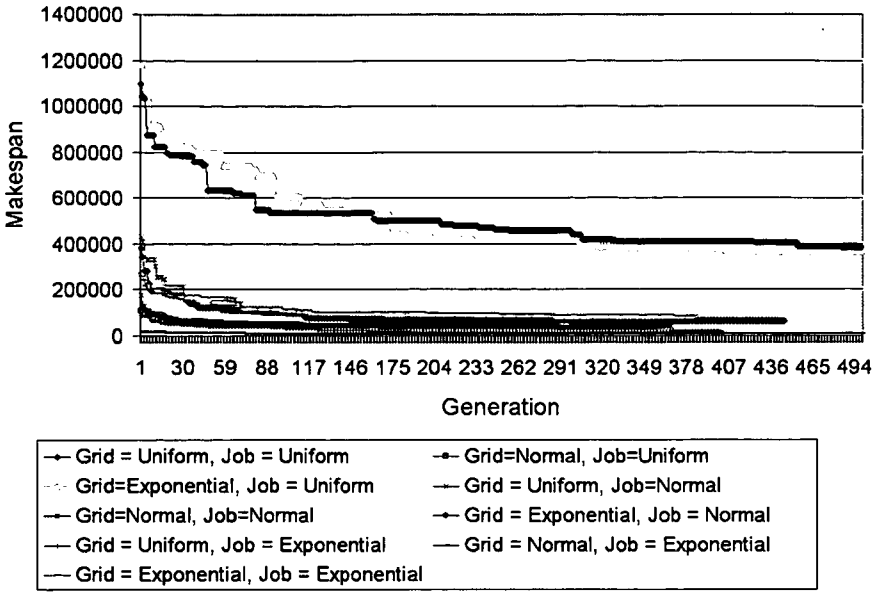


Fig. 7.28. Evolution of the best individual (makespan) across generations.

8

Related Work

The work presented in this thesis is centred around four different research areas: scientific experiment management, performance studies, parameter studies, and scheduling as an optimisation task. The most relevant related work in each of these areas will be outlined in separate sections of this chapter.

8.1 Experiment Management

The ZOO project [93] has been initiated to support scientific experiment management based on a desktop environment. A clear lifecycle of a scientific experiment has been defined, which iterates through three steps: experiment design, data collection, and data exploration. Experiments are designed by using an object-oriented data description language. Input data are provided through a special experiment database. A transformation mechanism maps the contents of the database to application specific input and output files. In contrast, ZENTURIO and the ZEN language do not restrict the parametrisation to input files, but enable the parameter specification within arbitrary application files.

The Unicore project (Uniform Interface for COmputing REsources) [136] facilitates the usage of supercomputers on the Grid by using modern browser technology. Experiments have to be manually set-up, including source, input, and output file staging, as well as and eventual compilation processes. There is no support for experiment set specification and automatic experiment management. Performance analysis is supported for single experiments by providing an interface to the Vampir performance tool [117]. The Unicore jobs can be manually organised in a directed acyclic graph comprising file transfers and binary or script file executions (including compilation and link tasks). The workflow jobs can be organised in groups. Automatic scheduling is not addressed.

8.2 Performance Study

The Paradyn parallel performance tools [95] supports experiment management through a representation of the execution space of performance experiments, techniques for the quantitative comparison of several experiments, and performance diagnosis based on dynamic instrumentation. The experiments have to be set up manually, whereas performance analysis is done automatically for every experiment based on historical data harnessing [96]. Paradyn is based on dynamic instrumentation which is difficult to apply for high-level programming paradigms like OpenMP and HPF. In contrast, the ZEN performance directives of ZENTURIO support compile-time instrumentation of arbitrary source code regions and high-level language-specific performance overheads.

The National Institute of Standards and Technology (NIST) [36] developed a prototype for an automated benchmarking tool-set to reduce the manual effort in running and analysing the results of parallel benchmarks. A data collection and storage module implements a central repository for the collection of performance data. A visualisation module provides an integrated mechanism to analyse and visualise the data stored in the repository. An experiment control module assists the user in designing and executing the experiments. In contrast to ZENTURIO, the experiment specification is restricted to pre-defined parameters available through a special purpose graphical user interface.

SKaMPI [135] provides a benchmarking environment for MPI applications with the goal of analysing the runtime of the MPI routines. A pre-defined set of measurements, machine, and problem size parameters can be controlled by the programmer through a special-purpose planning script. A public performance database allows the storage of the benchmark data and the interactive comparison of various MPI performance aspects across different implementations and platforms. The project, however, focuses exclusively on benchmarking various MPI implementations.

The Tracefile Testbed [60] is a new community repository for organising the performance data of parallel applications. It allows the users to flexibly search and retrieve the trace file metadata based on specific parameters such as the computer platform used, the types of events recorded, or the class of applications. The automatic execution of experiments and the automatic data collection are not addressed.

The XPARE (eXPeriment Alerting and REporting) [43] tools are designed to specify automated benchmark regression testings for a given set of performance measurements of parallel applications. A historical panorama of the performance metric evolution across software versions is provided. Apart from software versioning, no other parametrisation is addressed.

The IST APART working group developed as part of Workpackage 3 [115] a generic design of an automatic performance analysis system that defines and categorises the performance analysis experiments.

Automatically Tuned Linear Algebra Software (ATLAS) [174] is an empirical approach for automatic generation and optimisation of numerical software for processors with deep memory hierarchies and pipelined functional units. Benchmarking data is organised and stored in a special-purpose *Performance Database Server (PDS)* [17]. The scope of ATLAS is, however, limited to linear algebra software and comprises a pre-defined set of parameters and optimisation metrics.

8.3 Parameter Study

Nimrod [2] is a tool that manages the execution of parameter studies across distributed computers by hiding the low-level issues of distributing files to remote systems, performing remote computations, and gathering results. A parameterised experiment is specified by a declarative plan file which describes the parameters, their default values, and the commands necessary for performing the work. Nimrod generates one job for each unique combination of parameter values, by taking the cross product of all the values. The set of possible parameter value combinations cannot be constrained. As a limitation to the ZENTURIO approach, the parameterisation is restricted to global variables which requires appropriate adaptation of the application. Remote source code compilation is not addressed. Other research prototypes of the tool include application specific interfaces for controlling parameters.

The ILAB [178] project controls parameter studies through graphical annotations of input files. Value sets can be specified by enumeration lists or by *min:max:inc* patterns. Masking of parameter values is supported via PERL scripts. Program variables cannot be controlled.

8.4 Optimisation and Scheduling

The Directed Acyclic Graph Manager (DAGMan) [154] is a meta-scheduler for Condor [108]. DAGMan manages I/O data and control dependencies between jobs at a higher level than the Condor scheduler. The DAG is specified by a special input script where each node is described by a Condor submit description file. The concrete scheduling is based on Condor specific techniques such as resource matchmaking and cycle stealing. Recursive workflow loops are not supported.

The Pegasus [44] system advocates (but not yet implements) Artificial Intelligence planning techniques to approach the workflow scheduling problem. Workflows are restricted to DAGs based on the Condor DAGMan model. Large workflows are reduced to more manageable quantities based on the Chimera virtual data [69] availability. Heuristics for optimising workflow schedules ahead of time are not considered. Rather, the workflow tasks are scheduled randomly to the Grid sites where the virtual data is available.

The AppLeS Parameter Sweep Template (APST) [29] uses the application-level scheduling techniques developed by the AppLeS [16] project for efficient deployment of parameter sweep applications over the Grid. The throughput optimisation algorithms comprise Min-Min, Max-Min, and Sufferage heuristics [112].

The GrADS project [41] continues the tradition of the AppLeS effort on developing techniques for scheduling MPI, iterative, and master-worker applications on the Grid, with recent focus on DAG-based workflows [53]. Unlike in ZENTURIO, DG-based workflow loops are not addressed. The static scheduling is approached through Max-min, Min-min, and Suffrage heuristics, typically used for throughput scheduling of independent tasks. Experimental results with ZENTURIO [129] prove that the complexity of classical Max-min-like heuristics are typically one order of magnitude higher than genetic algorithms ($\mathcal{O}(n^3)$ versus $\mathcal{O}(n^2)$). In [177], a simulated annealing algorithm for static scheduling of ScaLAPACK MPI applications has been successfully applied.

GridFlow [28] comprises a user portal and a set of services for global Grid workflow management and local Grid sub-workflow scheduling. Simulation, execution, and monitoring functionalities are provided at the global Grid level on top of an existing agent-based Grid resource management system. At each local Grid, sub-workflow scheduling and conflict management are processed on top of an existing performance prediction based task scheduling system. A fuzzy timing technique is applied for workflow management in a cross-domain and highly dynamic Grid environment.

Nimrod/O [4] is a variation of the Nimrod parameter study tool that uses a broad-range of heuristics for output parameter optimisations. Performance-oriented optimisations are not addressed and genetic algorithms are not used.

Nimrod/G [3] is a Grid-aware version of Nimrod enhanced with ad-hoc techniques for throughput scheduling of parameter studies on multiple Grid sites based on a user-defined budget and deadline functionality. The Nimrod/G scheduler is based on a computational economy model called *GRACE (GRid Architecture for Computational Economy)* and does not target general NP-complete optimisations.

The problem of scheduling task graphs through genetic algorithms has been addressed in the past [104], however, restricted to homogeneous parallel computers with limited number of processors.

In [143], a hierarchical genetic algorithm has been successfully applied for automatic optimisation of HPF array distributions within Fortran 90 compilers. The definition of the objective function is based on training set pre-measurements.

8.5 Tool Integration

The Annai [33] tool environment has been the outcome of the collaboration between the Swiss Centre for Scientific Computing (CSCS) and NEC in developing an integrated parallel application engineering environment for parallel processing. Annai consists of an extended HPF compiler, a parallel performance monitor and analyser, and a parallel debugger for distributed memory parallel processors. While integration of different tools is achieved by the specification of well-defined interfaces and communication protocols, further extensions are only possible after rethinking, redesigning, and rebuilding the whole system.

The Portable Parallel Distributed Debugger (p2d2) [86] developed by NASA Ames Research Center promotes the idea of client-server tools, with platform dependencies confined to the server back-end, and the client front-end implemented in a portable manner. The debugger defines a server interface that should be implemented by any vendor which allows third party front-end clients be implemented in a platform independent manner.

The Tool-Set [111] integrated tool environment and the *On-line Monitoring Interface Specification (OMIS)* [110], both developed at the Technical University of Munich, build on the ideas of p2d2 affirming that a monitoring system should separate the application processes from the tools, thereby encapsulating the platform dependencies. OMIS defines an open interface for connecting run-time development tools in a distributed environment with the tool interoperability as a major requirement. Neither p2d2 nor OMIS, however, build their ideas on top of modern Grid technologies such as Web services.

DDBG/PDBG/TDBG [37] developed at the University Nova of Lisbon is a suite of distributed debuggers integrated into a wider-scope problem solving environment. DDBG has been interfaced to a graphical parallel programming tool for high-level debugging of parallel programs, and a static analysis and testing tool for controlled execution of previously generated testing scenarios.

The Parallel Tool Consortium (PTools) coordinated projects in the late 1990s with the purpose to define, develop, and promote parallel tools for scalable portable applications. These tools provide flexible open interfaces which facilitate their integration and reuse, however, the possibility of integration and interoperability has not been addressed.

The High Performance Debugging Forum (HPDF) [105] has defined within the PTools umbrella a useful and appropriate set of standards relevant to debugging tool development for high-performance computers.

The Dyninst [24] library developed at the University of Maryland exports a platform independent API to the dynamic instrumentation technology provided by the Paradyn project for portable dynamic instrumentation of single processes.

The Dynamic Probe Class Library (DPCL) [85] is an object based C++ class library that provides the tool developers with an advanced infrastructure

for building parallel and serial tools based on the dynamic instrumentation technology. DPCL allows the tool researchers focus on developing tools rather than deal with compiler details or distributed infrastructure development.

9

Conclusions

9.1 Contributions

This section concludes the thesis by summarising the main contributions in the areas of experiment management, optimisation, and tool integration in Grid computing. In addition, it gives an outlook to a series of potential future research directions.

9.1.1 Experiment Specification

A new *directive-based language* called ZEN [123, 127] has been designed to specify the set of experiments which is subject to potentially large performance, parameter, or optimisation studies. The so called ZEN directives are program comments that annotate arbitrary application files and, therefore, do not change the semantics of the code, as they are ignored by compilers or interpreters which are not aware of their semantics.

The ZEN directives defined by the ZEN language can be summarised as follows:

1. *Substitute directives* allow flexible specification of arbitrary application parameters through string substitution semantics. The ZEN substitute directives are useful for defining application parameters beyond ordinary program variables, like array distributions, loop scheduling strategies, file paths, compiler options, or target machines;
2. *Assignment directives* are used to parameterise program variables in cases when the substitute directives are inconvenient or impossible to be used. A typical case for using the assignment directives are the parameterisations of variable with short names (e.g., `N`) for which the substitute directives would also replace other equal but invalid string occurrences (e.g., in key words like `END`);
3. *Constraint directives* are used to restrict the overall number of experiments to a meaningful subset;

4. *Performance directives* are used to the specify the high-level performance metrics (i.e., OpenMP, MPI, and HPF-specific) to be measured and computed for fine-grained code regions, without altering the application source code with instrumentation probes.

The scope of the ZEN directives can be global or restricted to arbitrary code regions.

The ZEN directive-based language presents the following advantages against other existing ad-hoc scripting [2] or graphical [178] parameter specification alternatives:

1. it does not require special preparation of the application, which is an essential feature of a tool to achieve general acceptance;
2. it does not restrict the parametrisation to global variables to be exported outside the scope of the source code;
3. it can parameterise arbitrary local variables with arbitrary (even equal) names;
4. it can parameterise arbitrary application characteristics (e.g., parallelisation options) like array and loop distributions, software libraries, problem and machine sizes, target execution machines, communication networks, or compilation options;
5. it can apply at arbitrary fine-grained scopes within the application source files.

The thesis illustrated a variety of real-world scenarios [122, 130] how a wide set (e.g., thousands) of experiments can be expressed through a small number (e.g., under 10) of short (e.g., under 50 characters) ZEN directives.

9.1.2 Experiment Management

The thesis proposes a general-purpose *experiment management tool* called ZENTURIO [124, 128] applied for *cross-experiment performance and parameter studies* of parallel and Grid applications. ZENTURIO employs the ZEN directive-based language to define wide value ranges for arbitrary application parameters, including program variables, file names, compiler options, target machines, machine sizes, scheduling strategies, or data distributions, without altering the source code or requiring any application modification. A lightweight graphical User Portal easy to be installed and managed enables the user to create, control, and monitor the experiments as they progress from arbitrary Grid locations (i.e., client sites). After the manual annotation of the application with ZEN directives, ZENTURIO automatically generates and conducts the complete set of experiments. Upon the completion of each experiment, the performance and output data are automatically stored into a relational Experiment Data Repository for post-mortem analysis. An Application Data Visualiser portlet is used to automatically query the repository and visualise the cross-experiment variation of any performance metric or

output parameter as a function of arbitrary application parameters (i.e., ZEN variables).

The multi-experimental performance analysis automatically performed by ZENTURIO is a unique contributing research feature. The parameter study support within ZENTURIO is a low hanging side-effect fruit of the general experiment specification approach taken by the ZEN language.

The practical usefulness of ZENTURIO as a performance and parameter study tool has been demonstrated on a variety real-world parallel applications [130]. Additionally, ZENTURIO has been installed and used for benchmarking three-dimensional FFT kernels at the Paul Scherrer Institute (Swiss Federal Institute of Technology – ETH Zurich) in Switzerland as part of an international cooperation [122].

9.1.3 Optimisation

ZENTURIO proposes a novel *optimisation framework* [129] that employs general-purpose heuristic algorithms to reduce the parameter space defined through ZEN directives while searching for experiments that optimise a certain output parameter or a performance metric. The framework can be easily instantiated for a wide variety of performance and parameter optimisation problems by simply supplying the objective function to be maximised. The platform dependency of the objective function is hidden under problem independent interface. ZENTURIO illustrates a generic encoding of the optimisation search engine based on genetic algorithms and is open to other heuristics.

The following framework instantiations, all novel in the Grid computing arena, have been illustrated as case studies:

1. *Static scheduling of single workflow applications on the Grid* using genetic algorithms has been applied for the first time and demonstrated to provide effective high-quality results. This feature has been successfully applied on a real-world material science workflow application;
2. *Throughput scheduling of large sets of independent tasks on the Grid* using genetic algorithms has been demonstrated to have the potential of converging to good results by visiting a number of search space points of an order of magnitude lower than classical Max-min algorithms;
3. *Optimisations of parallel applications* by repeated experimentation using a genetic search engine is a novel technique. A modular design enables to supply the performance metric to be optimised by means of ZEN performance directives. A concrete instantiation for load balancing HPF applications on heterogeneous Grid resources using irregular array distributions has been formally described.

9.1.4 Dynamic Workflow Scheduling

The thesis proposes a novel hybrid algorithm for *dynamic scheduling* of workflow applications on the Grid. The contributions of the new algorithm can be summarised as follows:

1. it iteratively applies the static scheduling algorithm for optimised mapping of entire DAG-based workflows on the Grid;
2. it defines a set of cycle elimination rules for run-time conversion of DG-based workflows into DAG-based workflows, which are the subject to optimised static scheduling;
3. it defines rules for workflow task migration based on performance contract violation;
4. simulated results of a real-world material science workflow execution demonstrates that the algorithm outperforms existing lazy just-in-time or random workflow scheduling approaches.

9.1.5 Tool Integration Design

The ZENTURIO experiment management tool has been designed within a broad *tool integration framework for interoperability* which brings the following design contributions [97, 98]:

1. *A layered architecture* that isolates the platform dependencies under a portable API significantly increases the tool availability and portability. The recommendation that each vendor provide the required set of platform dependent sensors (and eventual services) under a platform independent API significantly increases the cross-platform tool availability and therefore, the acceptance of new computing platforms in the user community;
2. *The hardware and operating system dependencies* which are inherent to run-time tool development are insulated within sensors exporting a portable interface;
3. *A broad set of high-level services and sensors* that eases the portable tool development has been designed and implemented:
   a) A *Process Manager* sensor encapsulates the platform dependencies for manipulation and dynamic run-time instrumentation of single processes;
   b) *Experiment Generator service* encapsulates the platform dependencies (including proprietary software libraries) of the Vienna Fortran Compiler, on which the implementation of the ZEN performance directive is based;
   c) *Experiment Executor* is a general purpose service for remote execution and management of experiments on the Grid interfaced to a variety of batch job schedulers;
   d) *Dynamic Instrumentor* service exports a platform independent interface for low-level process management, on-the-fly run-time dynamic instrumentation, and on-line performance data collection;

4. *The concurrent service access* by multiple clients enables end-user tools interoperate through the common use of services;
5. *Light-weight clients* or end-user tools which are easy to be installed and managed are promoted. The client tool functionality is built through the concurrent use of the underlying high-level Grid services;
6. *Tool interoperability* [98, 132] has been classified and various scenarios that can improve the application engineering process have been proposed and prototyped. In this context, an SQL-based relational *Experiment Data Repository* that enables post-mortem performance and output data exchange has been designed.

9.1.6 Web Services for the Grid

The thesis contributes with techniques regarding the use of the *Web services technology* for *modelling stateful Grid resources*, which anticipated several standardisation efforts currently still under way within the Global Grid Forum [125, 128].

1. *Factory* is a general purpose service for creating Grid service instances on remote Grid sites;
2. *Registry* is a general purpose service for high-throughput service discovery based on white, yellow, and green pages lookup operations;
3. *The WSDL compatibility* for testing whether two Grid services implement the same functionality required for green pages lookup operations;
4. *The UDDI standard* for publishing persistent Web services has been redesigned for accommodating transient Grid services implementations;
5. *Service lifetime* has been addressed based on the extensions provided by existing Web services hosting environments;
6. *An event framework* compliant with the Grid Monitoring Architecture, but not restricted to performance analysis, has been implemented based on the Web services technology;
7. The emergence of new Grid standards has been continuously monitored and comparatively evaluated against the own infrastructure, which gave useful feedback to the community [126, 128].

9.2 Future Research

The following potential research directions are currently being considered for future research:

1. *New heuristics* besides genetic algorithms for general-purpose optimisations will be comparatively studied, including subdivision, simplex, simulated annealing, BFGS, or EPSOC methods;
2. *Various optimisations of parallel applications* on cluster and Grid architectures will be experimented, including parallelisation and scheduling;

3. *The static workflow scheduling* algorithm will be studied for the optimisation of other workflow metrics, in particular the efficiency;
4. *A meta-scheduler* for high-throughput scheduling of multiple workflows will be developed;
5. *Novel knowledge-based workflow specifications* (currently addressed by a parallel project at the University of Innsbruck) based on the semantic Web and ontology infrastructures [42] will be targeted for efficient scheduling;
6. *A language and platform independent workflow intermediate representation* that can be interfaced to multiple front-end (e.g., XML) workflow representations and back-end enactment engines (e.g., Globus CoG [7], Condor DAGMan [154]) will be studied. This will allow the optimisation and the scheduling techniques apply at a more abstract level (similar to the compiler abstract syntax tree intermediate representation) which is decoupled from the actual workflow representation and implementation;
7. *Fine-grained performance analysis* of single and multiple workflow applications will be studied and more appropriate Grid performance metrics will be formulated;
8. *The dynamic scheduling* problem will be addressed for other classes of applications, like parallel applications and parameter studies;
9. *The fault tolerance* will be addressed for all classes of Grid applications with particular interest for workflow enactment engines;
10. *Self-installing and self-healing* issues advocated by the autonomic computing vision [87] will be addressed for Grid applications;
11. *Collaborative applications* will be studied as a potentially new class of applications that could benefit of the techniques developed in this thesis.

10

Appendix

10.1 Notations

Symbol	Description
$\mathbb{N}$	Set of natural numbers
$\mathbb{N}^*$	Set of positive natural numbers (non-zero)
$\mathbb{R}$	Set of real numbers
$\mathbb{R}_+$	Set of positive real numbers
$[a, b]$	Set of real numbers from a to b
$[a..b]$	Set of integer numbers from a to b
$\Longleftrightarrow$	If and only if (iff)
$\Longrightarrow$	Implication
$\rightarrow$	Function mapping
$\forall$	For all
$\exists$	Exists
true	True boolean value
false	False boolean value
$\mid$	Set restriction
$\in$	Set membership
ϕ	Empty set
$\|S\|$	Cardinality of set S
$\mathcal{P}(S)$	Power set of S
$\times$	Cross product
$\Pi_P S$	Projection operator from space S to subspace P
$\wedge$	Logical conjunction
$\vee$	Logical disjunction
$\cup$	Set union
$\cap$	Set intersection
$\setminus$	Set difference
$\subset$	Subset of
$\prec$	Totally ordered set precedence

Symbol	Description
d	ZEN directive
z	ZEN variable
$\mathcal{S}$	ZEN set
e	ZEN element
ε	ZEN set evaluation function
$\mathcal{V}^z$	Value set of ZEN variable z
$\mathcal{I}^z$	Index domain of ZEN variable z
ϑ	Value function
ϑ^{-1}	Index function
$\tau(z)$	Type of ZEN variable z
$\nu(z)$	Name of ZEN variable z
$scope(d)$	Scope of ZEN directive d
γ	ZEN constraint function
$\otimes$	Tuple composition
$\mathcal{M}$	Performance measurement
$\mathcal{CR}$	Code Region
δ	Performance data
$\mathcal{OP}$	Output parameter
ϵ	Output data
$\mathcal{Z}$	ZEN file
$\mathcal{ZI}$	ZEN file instance
$\mathcal{Z}_o$	Output file
$\mathcal{A}$	ZEN application
$\mathcal{AI}$	ZEN application instance
ϵ	Experiment
$Nodes$	Set of graph nodes
$Edges$	Set of graph edges
N	Graph node or task
$pred(N)$	Predecessor of graph node N
$succ(N)$	Successor of graph node N
$pred^p(N)$	Predecessor of rank p of graph node N
$succ^p(N)$	Successor of rank p of graph node N
(N_1, N_2)	Directed graph edge from node N_1 to node N_2
ρ	Graph path
JS	Job submission task
FT	File transfer task
$Nodes^{JS}$	Set of JS tasks
$Nodes^{FT}$	Set of FT tasks
$\mathcal{G}$	Gantt chart
$\mathcal{S}_N$	Schedule of task N
t	Timestamp
$\mathcal{C}$	Constant
$start(N)$	Start timestamp of task N

Symbol	Description
$end(N)$	Termination timestamp of task N
$state(N, t)$	State of task N at timestamp t
flops	Floating point operations
W	Work
v	Speed
T	Execution time
$\mathcal{F}$	Objective or fitness function
$\overline{\mathcal{F}}$	Average population fitness
$\mathcal{P}$	Population of ZEN application instances
$\oplus$	Crossover operator
$\ominus$	Mutation operator
$PC(N, \mathcal{S}_N, t)$	Performance contract of task N with the schedule $\mathcal{S}_N$ at time instance t
f_N	Performance contract elapse factor of task N
$\mathcal{L}$	Latency
$\mathcal{B}$	Bandwidth
$\mathcal{O}(m^n)$	Algorithm complexity of m^n

References

1. D. Abramson, R. Giddy, and L. Kotler. High performance parametric modeling with nimrod/G: Killer application for the global grid? In *Proceedings of the 14th International Conference on Parallel and Distributed Processing Symposium (IPDPS-00)*, pages 520–528, Los Alamitos, May 2000. IEEE.
2. D. Abramson, R. Sosic, R. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Proceedings of the 4th IEEE Symposium on High Performance Distributed Computing (HPDC-95)*, pages 520–528, Virginia, August 1995. IEEE Computer Society Press.
3. David Abramson, Rajkumar Buyya, and Jonathan Giddy. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, October 2002.
4. David Abramson, Andrew Lewis, Tom Peachey, and Clive Fletcher. An automatic design optimization tool and its application to computational fluid dynamics. In ACM, editor, *SC2001: High Performance Networking and Computing. Denver, CO, November 10–16, 2001*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2001. ACM Press and IEEE Computer Society Press.
5. Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, May 2002.
6. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference*, pages 483–485, 1967.
7. Kaizar Amin, Mihael Hategan, Gregor von Laszewski, and Nester Zaluzec. Abstracting the Grid. In *12th Euromicro Conference on Parallel Distributed and Network based Processing (PDP 2004)*. IEEE Computer Society Press, A Corunã 2004.
8. Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Siebel Systems, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services (bpel4ws). Specification version 1.1, Microsoft, BEA, and IBM, May 2003.
9. Bob Atkinson, Giovanni Della-Libera, Satoshi Hada, Maryann Hondo, Phillip Hallam-Baker, Johannes Klein, Brian LaMacchia, Paul Leach, John Manfer-

delli, Hiroshi Maruyama, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, John Shewchuk, and Dan Simon. Web Services Security (WS-Security). Specification, Microsoft Corporation, April 2002.

10. 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.
11. E. Bayeh. The WebSphere Application Server architecture and programming model. *IBM Systems Journal*, 37(3):336–348, 1998.
12. R. Bell, A. D. Malony, , and S. Shende. A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *9th International Europar Conference(EuroPar 2003)*, Lecture Notes in Computer Science, Klagenfurt, Austria, August 2003. Springer-Verlag.
13. Messaoud Benantar. *Introduction to the Public Key Infrastructure for the Internet.* P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 2002.
14. S. Benkner. HPF+: High Performance Fortran for advanced industrial applications. *Lecture Notes in Computer Science*, 1401, 1998.
15. S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming, IOS Press, The Netherlands*, 7(1):67–81, 1999.
16. Francine D. Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *CD-ROM Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996. IEEE.
17. Michael W. Berry, Jack J. Dongarra, Brian H. LaRose, and Todd A. Letsche. PDS: a performance database server. *Scientific Programming*, 3(2):147–156, Summer 1994.
18. Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 78–88, Atlanta, GA, May 1999. ACM Press.
19. Dimple Bhatia, Vanco Burzevski, Maja Camuseva, Geoffrey Fox, Wojtek Furmanski, and Girish Premchandran. WebFlow — a visual programming paradigm for Web/Java based coarse grain distributed computing. *Concurrency: Practice and Experience*, 9(6):555–577, June 1997.
20. Peter Blaha, Karlheinz Schwarz, and Joachim Luitz. *WIEN97: A Full Potential Linearized Augmented Plane Wave Package for Calculating Crystal Properties.* Institute of Physical and Theoretical Chemistry, April 2000.
21. Nat Brown and Charlie Kindel. *Distributed Component Object Model protocol: DCOM/1.0.* Microsoft Corporation and Redmond, WA, January 1998.
22. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings SC'2000*, November 2000.
23. M. Bubak, W. Funika, B. Baliś, and R. Wismüller. Performance Measurement Support for MPI Applications with PATOP. In *Proc. PARA 2000 Workshop on Applied Parallel Computing*, Bergen, Norway, June 2000.
24. Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

25. Doug Bunting, Martin Chapman, Oisin Hurley, Mark Little, Jeff Mischkinsky, Eric Newcomer, Jim Webber, and Keith Swenson. Web Services Context (WS-Context). Specification, Arjuna Technologies Ltd., Fujitsu Limited, IONA Technologies Ltd., Oracle Corporation, and Sun Microsystems, Inc., July 2003.
26. Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.
27. Brent Callaghan. *NFS Illustrated.* Addison-Wesley, Reading, MA, USA, 2000.
28. Junwei Cao, Stephen A. Jarvis, Subhash Saini!, and Graham R. Nudd. GridFlow: Workflow Management for Grid Computing. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid (CCGrid 2003))*, Tokyo, Japan, May 2003. IEEE Computer Sociery Press.
29. Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The AppLeS Parameter Sweep Template: User-level middleware for the grid. In ACM, editor, *SC2000: High Performance Networking and Computing. Dallas Convention Center, Dallas, TX, USA, November 4–10, 2000*, pages 75–76, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. ACM Press and IEEE Computer Society Press.
30. Charlie Catlett. Standards for Grid computing: Global Grid Forum. *Journal of Grid Computing*, 1(1):3–7, 2003.
31. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL), March 2001. http://www.w3.org/TR/wsdl.
32. J. Clark and S. J. DeRose (Eds). "XML Path Language (XPath) Version 1.0". W3C Recommendation, April 1999. http://www.w3.org/TR/xpath.
33. Christian Clémençon, Akiyoshi Endo, Josef Fritscher, Andreas Müller, Roland Rühl, and Brian J. N. Wylie. Annai: An integrated parallel programming environment for multicomputers. In Amr Zaky and Ted Lewis, editors, *Tools and Environments for Parallel and Distributed Systems*, volume 2 of *Kluwer International Series in Software Engineering*, chapter 2, pages 33–59. Kluwer Academic Publishers, February 1996.
34. Cluster Resources, Inc. Maui Scheduler. http://www.supercluster.org/maui/.
35. John Colgrave and Karsten Januszewski. Using WSDL in a UDDI Registry. UDDI specifications tc best practices process, OASIS, Nov 2003. http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v2.htm.
36. Michel Courson, Alan Mink, Guillaume Marcais, and Benjamin Traverse. An automated benchmarking toolset. In *HPCN Europe*, pages 497–506, 2000.
37. José C. Cunha, co João Louren and Tiago Ant ao. An Experiment in Tool Integration: the DDBG Parallel and Distributed Debugger. *Euromicro Journal of Systems Architecture*, 45(11):897–907, 1999.
38. 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.
39. Karl Czajkowski, Ian Foster, and Carl Kesselman. Co-allocation services for computational grids. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing.* IEEE Computer Society Press, 1999.

40. 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.
41. Holly Dail, Otto Sievert, Francine Berman, Henri Casanova, Asim YarKhan, Sathish Vadhiyar, Jack Dongarra, Chuang Liu, Lingyun Yang, Dave Angulo, and Ian Foster. Scheduling in the Grid Application Development Software Project. *Resource Management in the Grid*, 2003.
42. John Davies, Dieter Fensel, and Frank van Harmelen. *Towards the Semantic Web: Ontology-Driven Knowledge Management.* John Wiley & Sons, 2003.
43. J. Davison de St. Germain, Alan Morris, Steven G. Parker, Allen D. Malony, and Sameer Shende. Integrating performance analysis in the uintah software development cycle. In *Proceedings of the The fourth International Symposium on High Performance Computing (ISHPC-IV)*, pages 190–206, Kansai Science City, Japan, 2002.
44. Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–39, 2003.
45. L. P. Deutsch and J-L. Gailly. RFC 1950: ZLIB compressed data format specification version 3.3, May 1996.
46. 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.
47. Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, August 2003.
48. Matthew J. Duftler, Nirmal K. Mukhi, Aleksander Slominski, and Sanjiva Weerawarana. Web Services Invocation Framework (WSIF). In *Proceedings of the OOPSLA 2001 Workshop on Object-Oriented Web Services*, Tampa, Florida, USA, October 2001.
49. M. Dumas and A. Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *4th International Conference on UML, LNCS 2185*, Toronto, Canada, October 2001. Springer Verlag.
50. W. K. Edwards. Core Jini. *IEEE Micro*, 19(5):10–10, September/October 1999.
51. The Mind Electric. GLUE. http://www.themindelectric.com/glue/index.html.
52. Dietmar W. Erwin and David F. Snelling. UNICORE: A Grid computing environment. *Lecture Notes in Computer Science*, 2150, 2001.
53. Ken Kennedy et.al. New Grid Scheduling and Rescheduling Methods in the GrADS Project. In *International Parallel and Distributed Processing Symposium, Workshop for Next Generation Software*, Santa Fe, New Mexico, April 2004. IEEE Computer Society Press.
54. T. Fahringer. *Automatic Performance Prediction of Parallel Programs.* Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.
55. T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, and H.-L. Truong. ASKALON - A Programming Environment and Tool Set for Cluster and Grid Computing. www.par.univie.ac.at/project/askalon, Institute for Software Science, University of Vienna.

56. T. Fahringer, S. Pllana, and A. Villazon. A-GWL: Abstract Grid Workflow Language. In *International Conference on Computational Science. Programming Paradigms for Grids and Metacomputing Systems.*, Krakow, Poland, June 2004. Springer-Verlag.
57. Thomas Fahringer. ASKALON Visualization Diagrams. http://www.par.univie.ac.at/project/askalon/visualization/index.html.
58. Jim Farley, William Crawford, and David Flanagan. *Java Enterprise in a Nutshell.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, 2002.
59. John Feller. IBM Web Services ToolKit – A Showcase for emerging Web Services Technologies. http://www-4.ibm.com/software/solutions/webservices/wstk-info.html, April 2002.
60. Ken Ferschweiler, Mariacarla Calzarossa, Cherri Pancake, Daniele Tessera, and Dylan Keon. A community databank for performance tracefiles. In Y. Cotronis and J. Dongarra, editors, *Euro PVM/MPI*, pages 233–240. Springer-Verlag, 2001. Lect. Notes Comput. Sci. vol. 2131.
61. Steve Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steve Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375, Portland, OR, 5-8 August 1997.
62. I. Foster, D. Gannon, and H. Kishimoto. *The Open Grid Services Architecture.* The Global Grid Forum, November 2003. https://forge.gridforum.org/projects/ogsa-wg.
63. I. Foster and N. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of SC'98.* ACM Press, 1998.
64. 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.
65. I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure.* Morgan Kaufmann, 2004. 2nd edition.
66. I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proc. IOPADS'97*, pages 14–25. ACM Press, 1997.
67. Ian Foster, Jeffrey Frey, Steve Graham, Steve Tuecke, Karl Czajkowski, Don Ferguson, Frank Leymann, Martin Nally, Tony Storey, William Vambenepe, and Sanjiva Weerawarana. Modeling Stateful Resources with Web Services. Specification, Globus Alliance, Argonne National Laboratory, IBM, USC ISI, Hewlett-Packard, January 2004.
68. Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS-98)*, pages 83–92, New York, November 3–5 1998. ACM Press.
69. Ian Foster, Jens Vockler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *14th International Conference on Scientific and Statistical Database Management (SSDBM 2002)*, September 2002.
70. Apache Software Foundation. Apache Axis. http://ws.apache.org/axis.

71. Bob Friesenhahn. Autoconf makes for portable software — use of OS features and a freeware scripting utility solves application portability across various flavors of Unix. *BYTE Magazine*, 22(11):45–46, November 1997.
72. Matteo Frigo and Steven G. Johnson. benchFFT. http://www.fftw.org/benchfft/.
73. Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.
74. Michael R. Garey and David S. Johnson. *Computers and Intractability / A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, San Francisco, 1978.
75. M. Geissler. *Interaction of High Intensity Ultrashort Laser Pulses with Plasmas.* PhD thesis, Vienna University of Technology, 2001.
76. G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instumented communications library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, Tennessee, January 1992.
77. David E. Goldberg. *Genetic Algorithms in Search, Optimization 6 Machine Learning.* Reading. Addison-Wesley, Massachusetts, 1989.
78. K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web services architecture. *IBM Systems Journal*, 41(2):168–177, 2002.
79. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.
80. William Grosso. *Java RMI.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 2002.
81. Elliotte Rusty Harold. *XML: EXtensible Markup Language.* IDG Books, San Mateo, CA, USA, 1998.
82. Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, January 1992.
83. Rolf Herzog. PostgreSQL — the Linux of databases. *Linux Journal*, 46, February 1998.
84. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.
85. Jeffrey K. Hollingsworth, Luiz Derose, and Ted Hoover. The dynamic probe class library - an infrastructure for developing instrumentation for performance tools. In *15th International Parallel and Distributed Processing Symposium (IPDPS 2001).* IEEE Computer Society Press, April 2001.
86. Robert Hood. The *p2d2* Project: Building a Portable Distributed Debugger. In *Proceedings of 1st SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'96, Philadelphia, PA, USA).* ACM Press, May 1996.
87. Paul Horn. Autonomic computing: IBM's perspective on the state of information technology. Manifesto, IBM Research, October 2001.
88. High Performance Fortran Forum, High Performance Fortran Language Specification. Version 2.0.δ, Technical Report, Rice University, Houston, TX, Jannuary 1997.
89. Soonwook Hwang and Carl Kesselman. Grid workflow: A flexible failure handling framework for the grid. In *Proceedings of the 12th IEEE International*

Symposium on High Performance Distributed Computing (HPDC-12), pages 126–137, Seattle, WA, USA, June 2003. IEEE Computer Society Press.

90. Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 24(2):280–289, April 1977.
91. IBM Corporation. *Using and Administering LoadLeveler – Release 3.0*, 4 edition, August 1996. Document Number SC23-3989-00.
92. IT Innovation. Workflow enactment engine, October 2002. http://www.it-innovation.soton.ac.uk/mygrid/workflow/.
93. 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.
94. Alexandru Jugravu and Thomas Fahringer. JavaSymphony, a Programming Model for the Grid. In *PPGaMS 2004 at International Conference on Computational Science 2004-ICCS 2004*, Krakow, Poland, June 2004. Springer Verlag.
95. 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.
96. Karen L. Karavanic and Barton P. Miller. Improving online performance diagnosis by the use of historical performance data. In ACM, editor, *Proceedings of the SC'99 Conference*, Portland, Oregon, November 1999. ACM Press and IEEE Computer Society Press.
97. John M. Kewley and Radu Prodan. A Distributed Object-Oriented Framework for Tool Development. In Q. Li, D. Firesmith, R. Riehle, G. Pour, and B. Mayer, editors, *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 34)*, pages 353–62. IEEE Computer Society Press, July 2000.
98. John M. Kewley and Radu Prodan. Interoperable Performance and Debugging Tools using Dynamic Instrumentation. *Parallel and Distributed Computing Practices*, Volume 4, Number 3, Special Issue on Monitoring Systems and Tool Interoperability:245–260, September 2001.
99. Chris Kostick. IP masquerading with Linux. *Linux Journal*, 27, July 1996.
100. Heather Kreger. Web Services Conceptual Architecture (WSCA 1.0). Prepared for Sun Microsystems, Inc., IBM Software Group, May 2001. http://www-4.ibm.com/software/solutions/webservices/pdf/WSCA.pdf+.
101. Sriram Krishnan, Randall Bramley, Dennis Gannon, Madhusudhan Govindaraju, Rahul Indurkar, Aleksander Slominski, Benjamin Temko, Jay Alameda, Richard Alkire, Timothy Drews, and Eric Webb. The XCAT science portal. In *SC'2001 Conference CD*, Denver, November 2001. ACM SIGARCH/IEEE.
102. Sriram Krishnan, Patrick Wagstrom, and Gregor von Laszewski. GSFL : A Workflow Framework for Grid Services. Technical Report, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, U.S.A., July 2002.
103. Krishna Kunchithapadam and Barton P. Miller. Integrating a Debugger and a Performance Tool for Steering. In M.L. Simmons, A.H. Hayes, J.S. Brown, and D.A. Reed, editors, *Debugging and Performance Tools for Parallel Computing Systems*, pages 53–64. IEEE Computer Society Press, 1996.

104. Yu-Kwong Kwok and Ishfaq Ahmad. Efficient scheduling of arbitrary TAsk graphs to multiprocessors using a parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 47(1):58–77, 25 November 1997.
105. David LaFrance-Linden. Challenges in Designing an HPF Debugger. *DIGITAL Technical Journal*, 9(3), January 1998.
106. Reuven M. Lerner. At the forge: Server-side Java with Jakarta-Tomcat. *Linux Journal*, 84:50, 52–54, 56–58, April 2001.
107. David S. Linthicum. CORBA 2.0? *Open Computing*, 12(2), February 1995.
108. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, Washington, D.C., USA, June 1988. IEEE Computer Society Press.
109. Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.
110. Thomas Ludwig and Roland Wismüller. OMIS 2.0 – A Universal Interface for Monitoring Systems. In M. Bubak, J. Dongarra, and J. Wasniewski, editors, *Proceedings of 4th European PVM/MPI Users' Group Meeting*, pages 267–276. Springer Verlag, May 1997.
111. Thomas Ludwig, Roland Wismüller, Rolf Borgeest, Stefan Lamberts, Christian Röder, Georg Stellner, and Arndt Bode. THE TOOL-SET – An Integrated Tool Environment for PVM. In Jack Dongarra, M. Gengler, Bernard Touracheau, and Xavier Vigouroux, editors, *Proceedings of 2nd Euro PVM User's Group Meeting (EuroPVM'95, Lyon, France) Short Papers*. Ecole Normale Supérieure de Lyon, September 1995.
112. Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, November 1999.
113. A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington. Iceni dataflow and workflow: Composition and scheduling in space and time. In *UK e-Science All Hands Meeting*, pages 627–634, Nottingham, UK, September 2003.
114. Sun Microsystems. The Web Services Development Pack. http://java.sun.com/webservices/webservicespack.html.
115. B. Mohr. Design of Automatic Performance Analysis Systems, APART Workpackage 3: Implementation Issues. APART technical report, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik (ZMG), D-52425 Jülich, May 2000. http://www.kfa-juelich.de/apart.
116. MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical report, The Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee, jul 1997. `http://www.mpi-forum.org/`.
117. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.
118. Greg Nyberg. *WebLogic 6.1 Server Workbook for Enterprise JavaBeans*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, third edition, 2002.

119. Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org.
120. Andy Oram, editor. *Peer-to-peer: Harnessing the Power of Disruptive Technologies.* O'Reilly, Sebastopol, California, 2001.
121. S. Pllana, T. Fahringer, J. Testori, S. Benkner, and I. Brandic. Towards an UML Based Graphical Representation of Grid Workflow Applications. In *The 2nd European Across Grids Conference*, Nicosia, Cyprus, January 2004. Springer-Verlag.
122. Radu Prodan, Andreas Bonelli, Andreas Adelmann, Thomas Fahringer, and Christoph Überhuber. Benchmarking Parallel Three-Dimensional FFT Kernels with ZENTURIO. In *International Conference on Computational Science 2004 (ICCS 2004)*, volume 3037 of *Lecture Notes in Computer Science*, pages 459–467. Springer Verlag, June 2004.
123. Radu Prodan and Thomas Fahringer. ZEN: A Directive-based Language for Automatic Experiment Management of Parallel and Distributed Programs. In *Proceedings of the 31st International Conference on Parallel Processing (ICPP 2002)*. IEEE Computer Society Press, August 2002.
124. Radu Prodan and Thomas Fahringer. ZENTURIO: An Experiment Management System for Cluster and Grid Computing. In *Proceedings of the 4th International Conference on Cluster Computing (CLUSTER 2002)*. IEEE Computer Society Press, September 2002.
125. Radu Prodan and Thomas Fahringer. A Web Service-based Experiment Management System for the Grid. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003. IEEE Computer Society Press. Best Paper Award.
126. Radu Prodan and Thomas Fahringer. From Web Services to OGSA: Experiences in Implementing an OGSA-based Grid Application. In *4th International Workshop on Grid Computing (Grid 2003)*. IEEE Computer Society Press, November 2003.
127. Radu Prodan and Thomas Fahringer. ZEN: A Directive-based Experiment Specification Language for Performance and Parameter Studies of Parallel and Distributed Scientific Applications. *International Journal of High Performance Computing and Networking*, 2003. (To Appear).
128. Radu Prodan and Thomas Fahringer. ZENTURIO: A Grid Middleware-based Tool for Experiment Management of Parallel and Distributed Applications. *Journal of Parallel and Distributed Computing*, 64/6:693–707, 2004.
129. Radu Prodan and Thomas Fahringer. ZENTURIO: A Grid Service-based Tool for Optimising Parallel and Grid Applications. *Journal of Grid Computing*, 2004. To appear.
130. Radu Prodan, Thomas Fahringer, Franz Franchetti, Michael Geissler, Georg Madsen, and Hans Moritsch. On using ZENTURIO for Performance and Parameter Studies on Clusters and Grids. In *11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP 2003)*. IEEE Computer Society Press, February 2003.
131. Radu Prodan and John M. Kewley. FIRST: A Framework for Interoperable Resources, Services, and Tools. In H. R. Arabnia, editor, *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 4, pages 1790–96, Las Vegas, Nevada, USA, June 1999. CSREA Press.

132. Radu Prodan and John M. Kewley. A Framework for an Interoperable Tool Environment. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *EuroPar 2000, Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 65–69. Springer-Verlag, August 2000.
133. Rajesh Raman, Miron Livny, and Marvin Solomon. Policy driven heterogeneous resource co-allocation with gangmatching. In *Proceedings of the 12th IEEE Symposium on High Performance Distributed Computing (HPDC12)*, pages 80–89. IEEE Computer Society Press, 2003.
134. D. A. REED, R. A. AYDT, R. J. NOE, P. C. ROTH, K. A. SHIELDS, B. W. SCHWARTZ, and L. F. TAVERA. Scalable performance analysis: The pablo performance analysis environment. In A. Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.
135. R. Reussner, P. Sanders, L. Prechelt, and M. Mueller. SKaMPI: A detailed, accurate MPI benchmark. *Lecture Notes in Computer Science*, 1497:52, 1998.
136. 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.
137. M. Ronsse, K. De Bosschere, and Chassin de Kergommeaux. Execution Replay and Debugging. In M. Brugge M. Ducasse, editor, *Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG2000)*, pages 5–18, Munich, August 2000. TUM/IRISA, Computer Research Repository (CoRR).
138. M. Ronsse, K. De Bosschere, and Chassin de Kergommeaux. Non-intrusive on-the-fly data race detection using execution replay. In M. Brugge M. Ducasse, editor, *Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG2000)*, pages 148–163, Munich, August 2000. TUM/IRISA.
139. Ward Rosenberry and Jim Teague. *Distributing Applications Across DCE and Windows NT*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, November 1993.
140. Bill Roth. An introduction to Enterprise Java Beans technology. *Java Report: The Source for Java Development*, 3, October 1998.
141. 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.
142. Bjorn Satdeva. DHCP: The next generation host configuration scheme. *Sys Admin: The Journal for UNIX Systems Administrators*, 4(1), January/February 1995.
143. U. Nagaraj Shenoy, Y. N. Srikant, V. P. Bhatkar, and Sandeep Kohli. Automatic data partitioning by hierarchical genetic search. *Journal of Parallel Algorithms and Architecture*, 1999.
144. C. Sivula. A call for distributed computing (RPC). *Datamation*, 36(1):75–76, 78, 80, January 1990.
145. Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1996.
146. G. Stellner. CoCheck: checkpointing and process migration for MPI. In IEEE, editor, *Proceedings of IPPS '96. The 10th International Parallel Processing*

Symposium: Honolulu, HI, USA, 15–19 April 1996, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

147. Thomas L. Sterling and Hans P. Zima. Gilgamesh: A multithreaded processor-in-memory architecture for petaflops computing. In *SC'2002 Conference CD*, Baltimore, MD, November 2002. IEEE/ACM SIGARCH.
148. W. Richard Stevens and Stephen Rago. *Advanced Programming in the UNIX Environment.* Addison-Wesley, Reading, MA, USA, second edition, 2004.
149. Heinz Stockinger, Asad Samar, Bill Allcock, Ian Foster, Koen Holtman, and Brian Tierney. File and object replication in data grids. *Journal of Cluster Computing*, 5(3):305–314, 2002.
150. H.M. Stommel. The western intensification of wind-driven ocean currents. *Transactions American Geophysical Union*, 29:202–206, 1948.
151. Xian-He Sun and Diane T. Rover. Scalability of parallel algorithm-machine combinations. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):599–613, June 1994.
152. Sun Microsystems. Sun Grid Engine. http://gridengine.sunsource.net/.
153. Systinet. Web Applications and Services Platform for Java. http://www.systinet.com/doc/wasp_jserver/index0.html.
154. The Condor Team. Dagman (directed acyclic graph manager). http://www.cs.wisc.edu/condor/dagman/.
155. Clive Temperton. Self-sorting in-place fast Fourier transforms. *SIAM Journal on Scientific and Statistical Computing*, 12(4):808–823, July 1991.
156. Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.
157. Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany. *A Grid Monitoring Architecture.* The Global Grid Forum, January 2002. http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-16-2.pdf+.
158. Hong-Linh Truong and Thomas Fahringer. SCALEA Version 1.0: User's guide. Technical report, Institute for Software Science, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria, April 2001.
159. Hong-Linh Truong and Thomas Fahringer. SCALEA: A Performance Analysis Tool for Distributed and Parallel Program. In *8th International Europar Conference(EuroPar 2002)*, Lecture Notes in Computer Science, Paderborn, Germany, August 2002. Springer-Verlag.
160. Hong-Linh Truong and Thomas Fahringer. On Utilizing Experiment Data Repository for Performance Analysis of Parallel Applications. In *9th International Europar Conference(EuroPar 2003)*, Lecture Notes in Computer Science, Klagenfurt, Austria, August 2003. Springer-Verlag.
161. Hong-Linh Truong and Thomas Fahringer. SCALEA: A Performance Analysis Tool for Parallel Programs. *Concurrency and Computation: Practice and Experience*, 15(11-12):1001–1025, 2003.
162. Hong-Linh Truong and Thomas Fahringer. SCALEA-G: a Unified Monitoring and Performance Analysis System for the Grid. In *2nd European Across Grid Conference (AxGrids 2004)*, Lecture Notes in Computer Science, Nicosia, Cyprus, Jan 28-30 2004. Springer-Verlag.

163. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. *Open Grid Services Infrastructure.* The Global Grid Forum, June 2003. https://forge.gridforum.org/tracker/?atid=475&group_id=43&func=browse.
164. UDDI: Universal Description, Discovery and Integration. http://www.uddi.org.
165. J. Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.
166. Veridian Systems. PBS: The Portable Batch System. http://www.openpbs.org.
167. Gregor von Laszewski, Beulah Alunkal, Kaizar Amin, Shawn Hampton, and Sandeep Nijsure. GridAnt-Client-side Workflow Management with Ant. Whitepaper, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, U.S.A., July 2002.
168. Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):645–662, July/August 2001.
169. Fredrik Vraalsen, Ruth A. Aydt, Celso L. Mendes, and Daniel A. Reed. Performance contracts: Predicting and monitoring grid application behavior. *Lecture Notes in Computer Science*, 2242:154–166, 2001.
170. W3C. Web Services Activity. http://www.w3.org/2002/ws/.
171. W3C. XML Schemas: Datatypes. http://www.w3.org/TR/xmlschema-2/.
172. W3C. Web Services Architecture. http://www.w3.org/TR/2002/WD-ws-arch-20021114/, November 2002.
173. The Workflow Management Coalition. http://www.wfmc.org/.
174. R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In ACM, editor, *SC'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference: Orange County Convention Center, Orlando, Florida, USA, November 7–13, 1998*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1998. ACM Press and IEEE Computer Society Press. Best Paper Award for Systems.
175. R. Wismüller and T. Ludwig. THE TOOL-SET – An Integrated Tool Environment for PVM. In H. Lidell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *Proc. High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 1029–1030, Brussels, Belgium, April 1996. Springer-Verlag.
176. Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, October 1999.
177. Asim YarKhan and Jack J. Dongarra. Experiments with scheduling using simulated annealing in a Grid environment. *Lecture Notes in Computer Science*, 2536:232–244, 2002.
178. 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.
179. Songnian Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL,

December 1992. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.92.