Domain Engineering and Generic Programming for Parallel Scientific Computing

vorgelegt von
Diplom-Mathematiker
Jens Gerlach

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
– Dr.-Ing –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. rer. nat. Hans-Ulrich Heiß
Gutachter: Prof. Dr.-Ing. Stefan Jähnichen
Gutachter: Prof. Dr. Sergei Gorlatch

Tag der wissenschaftlichen Aussprache: 4. Juli 2002

Berlin 2002
D 83

For my parents Brigitte and Peter Gerlach

Acknowledgement

I want to express my gratitude to many people who supported me while I was working on this dissertation and developing the Janus framework.

First of all I want to thank my advisors Professor Stefan Jähnichen and Professor Sergei Gorlatch, both from Technische Universität Berlin, for their scientific assistance. As the director of Fraunhofer FIRST, Professor Jähnichen provided important feedback, constant support and a great research infrastructure. Professor Gorlatch discussed many parts of this dissertation with me and helped me to keep the focus of this work. I also want to thank Professor Wolfgang Schröder-Preikschat of University of Magdeburg for his invaluable assistance and very helpful critics when I was mapping out the scope of this work.

I started working on the subject of this dissertation during my stay at the Tsukuba Research Center of the Real World Computing Partnership, Japan. My department heads Mitsuhisa Sato and Yukata Ishikawa were very supportive and provided an outstanding research environment. I owe both of them and my other Japanese colleagues deep gratitude.

After returning from Japan, my project leader Dr. Hans-Werner Pohl played a unique role to foster my work. He supported my research in many ways and, not unlike Mephistopheles, he pushed me to finish my dissertation. I know no user of Janus who was more eager than Dr. Pohl to apply it to ever new application classes.

My colleague Uwe Der was also very helpful and solved many practical problems of the Janus software. The discussions with Friedrich Wilhelm Schröer and Dr. Jörg Nolte gave me a lot of insight into related fields. I also have to mention my long-standing friend and colleague Dr. Gerd Heber whose application of Janus in complex scenarios provided a lot of feedback and inspiration.

Berlin, March 2001

Abstract

The development of software for scientific applications that rest on dynamic or irregular meshes causes many problems because of the conflicting issues of flexibility and efficiency.

This dissertation addresses these problems in the following way. Firstly, it applies the ideas of *domain engineering* to the field of data-parallel applications in order to design reusable software products that accelerate the development of specific applications in this domain. It starts with an analysis of typical data-parallel applications and formulates general requirements on the components that can be used in this domain. Secondly, using the ideas of *generic programming* the *Janus* software architecture is defined.

The resulting conceptual framework and C++ template library Janus provides a flexible and extensible collection of efficient data structures and algorithms for a broad class of data-parallel applications. In particular, finite difference methods, (adaptive) finite element methods, and data-parallel graph algorithms are supported. An outstanding advantage of providing a generic C++ framework is that it provides *application-oriented* abstractions that achieve high performance without relying on language extension or non-standard compiler technology. The C++ template mechanism allows to plug user-defined types into the Janus data structures and algorithms. Moreover, Janus components can easily be combined with standard software packages of this field.

A portable implementation of Janus for distributed-memory architectures that utilizes the standard Message Passing Interface (MPI) is described. The expressiveness of Janus is proven by the implementation of several standard problems from the realm of data-parallel scientific applications. The performance of Janus is evaluated by comparing the Janus applications with those that use other state-of-the-art components. The examination of scalability on a high-performance Linux cluster system shows that Janus is on par with current scientific software.

Zusammenfassung

Die Entwicklung von Software für wissenschaftliche Anwendungen, die auf dynamischen oder irregulären Gittern beruhen, ist mit vielen Problemen verbunden, da hier so unterschiedliche Ziele wie hohe Leistung und Flexibilität miteinander vereinbart werden müssen.

Die vorliegende Dissertation geht diese Probleme folgendermaßen an: Zunächst werden die Ideen des *domain engineering* auf das Gebiet datenparalleler Anwendungen angewandt, um wiederverwendbare Softwareprodukte zu entwerfen, deren Benutzung die Entwicklung konkreter Softwaresysteme auf diesem Gebiet beschleunigt. Hierbei wird eine umfassende Analyse datenparalleler Anwendungen durchgeführt und es werden allgemeine Anforderungen an zu entwickelnde Komponenten formuliert. In einem zweiten Schritt wird auf der Grundlage der gewonnen Kenntnisse und unter Benutzung der Ideen des generischen Programmierens die Janus Softwarearchitektur entworfen und implementiert.

Das sich daraus ergebende konzeptionelle Gerüst und die C++-*template* Bibliothek Janus stellt eine flexible und erweiterbare Sammlung effizienter Datenstrukturen und Algorithmen für eine umfassende Klasse datenparalleler Anwendungen dar. Insbesondere werden finite Differenz- und finite Elementverfahren sowie datenparallele Graphalgorithmen unterstützt.

Ein herausragender Vorteil einer generischen C++ Bibliothek wie Janus ist, dass ihre anwendungsorientierten Abstraktionen eine hohe Leistung liefern und dabei weder von Spracherweiterungen noch von nicht allgemein verfügbaren Kompilationstechniken abhängen. Die Benutzung von C++-Templates bei der Implementierung von Janus macht es sehr einfach, nutzerdefinierte Datentypen in die Komponenten von Janus zu integrieren, ohne dass dabei die Effizienz leidet. Ein weiterer Vorteil von Janus ist, dass es sehr einfach mit bereits existierenden Softwarepaketen kooperieren kann.

Diese Dissertation beschreibt eine portable Implementierung von Janus für Architekturen mit verteiltem Speicher, die auf der standardisierten

Kommunikationsbibliothek MPI beruht. Die Ausdruckskraft von Janus wird an Hand der Implementierung typischer Anwendungen aus dem Bereich des datenparallelen wissenschaftlichen Rechnens nachgewiesen. Die Leistungsfähigkeit der Komponenten von Janus wird bewertet, indem Janus-Applikationen mit vergleichbaren Implementierungen, die auf anderen Ansätzen beruhen, verglichen werden. Die Untersuchungen zur Skalierbarkeit von Janus-Applikationen auf einem Linux Clustersystem zeigen, dass Janus auch in dieser Hinsicht hohe Anforderungen erfüllt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides the background, main motivations and goals of this dissertation. The contributions of this work are stated and an overview on how this dissertation is structured is given.

This dissertation has evolved out of the work of the author in several projects funded by the Real World Computing Partnerships (RWCP), Japan. The objective of the RWCP PROMISE[89] project, conducted at GMD (now Fraunhofer) FIRST, was to develop a programming environment for scientific application with the objective

Make parallel programming as easy as sequential programming.

An explicit goal was that adaptive finite element methods can be efficiently and conveniently expressed within this environment. Moreover, the environment should be highly portable and interoperable with existing languages, libraries, and tools. Finite element methods are used to solve many important and very demanding problems of science and engineering. To solve *adaptive* finite element methods on parallel architectures is a particular challenging task.

The author of this thesis started working in this field as a member of the PROMOTER project (a predecessor of PROMISE). At that time, the emphasis of the author's research was *requirement analysis* of data parallel applications[39, 41]. During his stay at the Tsukuba Research Center of RWCP in Japan, the author developed a small parallel C++ template library *Janus*[43, 44] that was targeted at adaptive finite element methods. This first version of Janus already incorporated the insights that were gained during the analysis of data parallel applications.

Based on this prototype, the *Janus conceptual framework* and a redesigned Janus template library were developed and later adopted as the core of the PROMISE environment[40]. The main reason for this decision was that Janus can handle not only regular and static data parallel applications but also irregular and dynamic ones. Thus, in the process of incorporating Janus into PROMISE it became clear that the conceptual framework of Janus goes far beyond finite element methods. Cellular automata, finite difference methods, and large classes of data parallel graph algorithms can be easily expressed using Janus abstractions.

1.1 Motivation and Goals

The task of developing software in the domain of high performance scientific computing has to deal with special requirements.

The most severe requirements are the stringent constraints regarding the efficiency of the data structures and algorithms used in scientific applications. Traditionally, the quest for high performance has been of such a paramount importance that modern software engineering concepts have entered the field only partially. This is true despite their potential to simplify multidisciplinary software development

In satisfying these performance constraints, developers often have to deal with different programming models. These models reflect different hardware architectures including distributed memory systems and shared memory systems and combinations of the two. Beside these dependences on different memory models on the whole, the deep memory hierarchies (such as registers and caches) can tremendously impact the overall performance of a scientific application. This causes application developers to design their software often in a hardware-oriented way. In general, developers of scientific codes are very reluctant to use overhead-prone software engineering principles or languages.

This tendency is strengthened by the fact that the software environments of high performance computing systems are often limited when compared with end-user workstations. This hinders the use of newer design and implementation techniques that might require additional tool support. The latter issue has somewhat eased with the rise of high performance computing systems that are based on clustering state-of-the-art workstations.

Algorithm development is a key activity in scientific computing. Multigrid methods[104] for the solution of large sparse systems of equations or the Barnes-Hut algorithm[8] are well known examples of how *efficient algorithms* have been devised during the last decades. Implementing these algorithms on high performance computing systems requires sophisticated data structures. In the realm of dense linear algebra, implementations of the *Basic Linear Algebra Subroutines* BLAS[31] standard are heavily used. For sparse systems that result from discretization of partial differential equations, only a draft standard[80] exists—resulting in a huge amount of incompatible data structures.

Over the years, it has been recognized that the complexity of multidisciplinary scientific simulations places demand of software engineering techniques that can deliver reusable and interoperable work products in this field. These efforts include the definition and implementation of expressive, portable, and efficient parallel programming models, such as the Message Passing Interface[97] (MPI) for distributed memory systems or OpenMP[102] for multi-platform shared-memory parallel programming.

The ISCOPE (*Scientific Computing in Object-Oriented Parallel Environments*) conference series that has been held since 1997 is exemplary in this respect. There are also internet-based forums for discussing various aspects of this field most notably

- Scientific Computing in Object-Oriented Languages[82],
- The Java Grande forum[47],
- The Common Component Architecture Forum[35].

The most prominent software engineering methods discussed by these communities in the context of scientific computing are *object-oriented design*, *generic programming*, and *component based development*. In particular, the ideas of generic programming have been recognized as a viable approach to develop interoperable and efficient software.

Nevertheless, generic programming as a mean to design software does not solve the problem of devising the useful abstractions. A much broader approach must be taken by carefully analyzing the *domain*, that is the area of knowledge, for which the scientific software is developed.

Domain engineering takes advantage of knowledge and software products that was obtained by developing systems in a specific "area of knowledge", that is, the *domain* under consideration. The emphasis is on *knowledge management* and *engineering reusable software*. In particular, the latter aspect brings domain engineering in close contact with Parnas' idea of *program families*. Knowledge management, on the other hand, requires a thoroughly and repeatedly performed analysis of the requirements to the systems in the domain.

1.2 Contributions

This dissertation covers a broad range of themes. After analyzing the state of the art of software engineering in the realm of scientific computing an analysis of the domain of data-parallel scientific applications is performed. Based on this analysis, a software architecture of data structures and algorithms is defined. Then the components of a C++ template library that implements the generic architecture are presented. It is explained how this library can be ported to different parallel architectures and how demanding data parallel operations can be written with it.

The most important contribution of this dissertation is the *conceptual framework* of Janus. The semantic and syntactic requirements of the three fundamental Janus concepts Domain[1], Relation, and Property Function were devised after a thorough analysis of data parallel applications. Domain formulates requirements for finite sets, Relation describes dependences of domain elements, and Property Function is an abstraction for attributes that are associated with domains or relations.

These abstractions allow the *unified treatment* of regular and irregular problems. Yet, the essential differences are not obliterated so that the efficient treatment of regular applications is not hindered by too general abstractions.

The main result of the domain analysis is that differences between regular and irregular problems can be treated as differences in the initialization process. The differences in initialization are addressed by so-called *one-phase* and *two-phase* data structures and algorithms. Two-phase data structures are particularly useful on distributed memory architectures because the explicit separation of insertions and access operations enables implementations that can efficiently and almost transparently deal with the difference of local and remote memory operations.

The next major contribution are the C++ template classes and functions that provide an extensible implementation of the Janus conceptual framework. The implementation of the Janus template library is very com-

[1]Note the difference between the term "domain" in the sense of an area of knowledge and the term Domain that defines abstractions for finite sets as they occur in data parallel applications

pact, consisting of less than 6000 lines of ISO-C++, and is highly portable. Special layers within Janus provide port packages for different computing platforms—including distributed-memory architectures, shared-memory systems, and purely sequential implementations.

The data structures and algorithms provides by Janus are not restricted to the solution of discretized partial differential equations. Parallel graph algorithms and cellular automata simulation are covered as well. Contrary to many matrix and graph libraries, the Janus conceptual framework and its template library explicitly supports an execution on distributed memory architectures.

The third major contribution of this dissertation is the evaluation of the conceptual framework and its C++ implementation by means of typical data parallel applications. This evaluation covers bott the expressiveness of the Janus approach and its performance in comparison with state-of-the-art solutions.

1.3 Organization of this Thesis

The following *Chapter 2* gives a a summary relating to the state of the art of *software engineering* in the realm of parallel scientific computing. The complexity of scientific software and the particular role played by *performance* characterizes this domain. The most notable design-methods that are considered in that chapter are *object-oriented design*, *generic programming*, and *component software*. For each of the design methods, one or more examples from the field of scientific computing are given. It is elaborated that generic programming is a particular promising approach to develop scientific software.

In *Chapter 3* an *analysis* of the domain of data-parallel scientific applications is undertaken. Several examples, including simple cellular-automata, finite difference and finite element methods for the approximate solution of discretized partial differential equations, mesh refinement, and parallel graph algorithms are considered. The result of this domain analysis is a list of characterizing properties of data parallel scientific applications.

Chapter 4 describes *Janus*—a generic software architecture for data-parallel applications. Based on the results of the domain analysis in Chapter 3, it defines a framework of *concepts*, that is, requirements for the semantics, syntax, and complexity of components that are well-suited for scientific computations. The most important ideas of Janus are the concepts Domain, Relation, and Property Function and the idea to use so-called *two-phase data structures* to represent irregular or dynamic sets and relations.

Chapter 5 presents the *Janus template library*—a set of generic C++ classes, algorithms, and utilities that implement the design requirements of the Janus concepts. The emphasis here is on the expressiveness and efficiency of the implementation, that is, there are very light-weight components for regular problems and more elaborated ones for irregular applications.

Chapter 6 discusses the portable implementation and configuration of the Janus components of Chapter 5 for different classes of parallel architectures. The *Janus Distributed Engine* (JADE) is introduced whose layered architecture provides a small but expressive set of communication and synchronization primitives that isolates lower level details of an underlying distributed-memory architecture. A port of JADE for the standard Message Passing Interface (MPI) is explained. Issues of shared-memory and strictly-sequential implementations are also discussed.

Chapter 7 applies the work products of the domain analysis, that is the Janus framework and library, to the implementation of applications in the domain of data parallel applications. Here again, typical examples with varying complexities are considered in detail. Various aspects of the performance of these applications are evaluated discussed in Chapter 8.

Chapter 9 concludes this thesis by discussing the Janus generic architecture in the context of similar work. It also provides an outlook of future work based on Janus.

Chapter 2

Software Engineering for Parallel Scientific Computing

In this chapter, the why and how modern software engineering methods have entered the domain of scientific computing will be investigate. The most prominent of these methods are *object-oriented design* §2.4, *generic programming* §2.6, and *component software* §2.8. The general ideas of these approaches and how they relate to this special field of software development will be presented. In particular, it is investigated how they address the issues of reusability and high performance. In one or another way, these software engineering approaches treat the problem of developing and maintaining a set of related programs. that is *program families.*

2.1 Software Quality

Software is produced to satisfy customer needs. As with any product, customers have various metrics for the quality of the software they employ. The following list presents important quality characteristics (and subordinated quality aspects) of software[58].

Functionality of software regards its *completeness*, *correctness*, *security*, *compatibility*, and *interoperability.*

Reliability of software refers to its *non-deficiency*, *error-tolerance*, and *availability.*

Usability of software regards its *understandability*, *ease of learning*, *operability*, and *communicativeness.*

Efficiency of software encompasses *time economy* and *resource economy.*

Maintainability of software regards its *correctability*, *expandability*, and *testability.*

Portability of software encompasses its *independence of hardware and other software*, *installability*, and last but not least *reusability.*

In Chapter 1 the outstanding role of *efficiency* as a quality feature for large scale scientific applications was pointed out. It was also discussed how this striving for efficiency conflicts with applying techniques and tools that can increase software quality.

Later in this chapter, several popular software engineering design methods are investigated with respect to their relation to the efficiency requirements in the field of scientific computing.

2.2 Program Families

The concept *program family* was introduced by Parnas who observed that varying applications and hardware demands inevitably mean that software exists in many versions. His classic definition reads:

> We consider a set of programs to constitute a *family*, whenever it is worthwhile to study programs from the set by *first* studying the common properties of the set and *then* determining the special properties of the individual family members.
>
> Parnas[84]

The insight is that if a designer carefully considers the commonalities and differences of the family members then the cost of development and maintenance of the programs will be reduced. The main reason for this is that members of a program family share basic design decisions and can use identical or similar resources[51]. Program families provide solutions that satisfy individual requirements and avoid performance deficiencies caused by services that are not necessary for a less demanding user[85].

Designing program families requires a good understanding of the problem *and* solution domains and is therefore closely related to *domain engineering* §2.3.

2.2.1 Incremental System Design

Commonality analysis and *variability analysis* are the basic activities for designing program families. Commonality analysis identifies basic abstractions that are shared by all members of a family. Variability analysis differentiates a family into individual members or subfamilies.

Program families are naturally represented as hierarchies that occur as the result of *incremental system design*[51]. In incremental system design, the fundamental abstractions form a *minimal basis* of the family. Step by step, this basis is enriched by *minimal extensions.* These extensions themselves act as a new minimal basis, or *virtual machine*, for higher level family members. This leads to a tree-like structure of the family where branches denote subfamilies.

2.2.2 Example: A Family of Container Abstractions

Figure 2.1 gives a simplified view on the family graph of container abstractions of the C++ standard library that is also known as Standard Template Library (STL)[99].

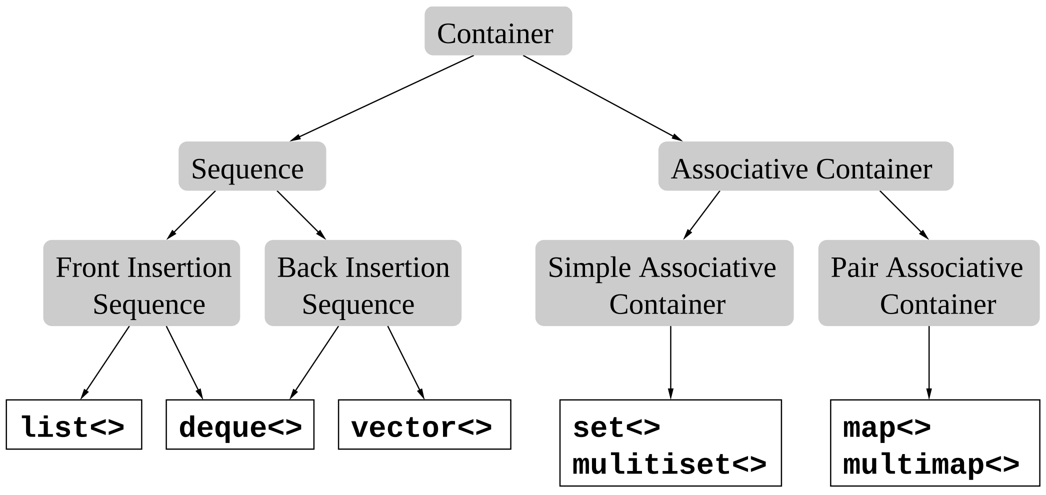

Figure 2.1: The STL container family

The shaded nodes represent incomplete members of the family, that is, they are not concrete software units but rather abstractions that define subfamilies. In this case, subfamilies are defined by *sets of requirements* that regard syntax, semantics, and complexity of the abstractions.

The minimal basis of the STL container family is the *Container* abstraction. In STL, the common properties of all containers are

- a container is an object that stores other objects (its elements), and that has methods for accessing its elements,
- a container must provide an associated iterator type that can be used to traverse the container's elements.

Subfamilies are introduced by extending the list of requirements of the preceding abstraction(s). The subfamily of *sequences*, for example, specifies that container elements are arranged in a strict linear order. It supports insertion of elements at a specific position. The subfamily of *associative containers*, on the other hand, supports efficient retrieval of elements (values) based on keys and does not provide a mechanism for inserting an element at a specific position.

Front insertion sequences form a subfamily of *sequences* that provide the methods `front`, `push_front`, and `pop_front` for accessing, inserting, and erasing, respectively, the first element of a sequence. An indispensable part of the semantics of these methods is that they have an amortized[1] constant complexity. Analogously, the family of *back insertion sequences* encompasses sequences where it is possible to append an element to the end, to access the last element, or to erase it in amortized constant time.

The "leaves" of this family graph are parameterized container types that are represented as C++ template classes. In contrast to the shaded nodes, the container template classes are directly usable software units. For example, the `list<T>` and `deque<T>` template container classes belong to the subfamily of front insertion sequences.

2.2.3 On the Implementation of Program Families

It is important to keep in mind that program families and incremental system design, in particular, are primarily *design* methods that do not prescribe particular implementation techniques. Habermann[51] explicitly states,

> It is the system design which is hierarchical, not its implementation.

[1] *Amortized complexity* means that the time required to perform a sequence of data structure operations is averaged over all the operations performed[25].

The reason is that individual family members must be in the position to fulfill special tasks without the overhead associated with unnecessary services.

The family of STL containers of 2.2.2 provides good examples for how the same design decision can be implemented in very different ways. The `list` and `deque` containers of Figure 2.1, for example, belong both to the subfamily of front insertion sequences. Whereas `list` is implemented as a doubly linked list, the implementation of `deque` typically rests on several contiguous memory blocks. This is because `deque` supports, in contrast to the `list` container, random access to its elements in amortized constant time. The `list` container, on the other hand, is the preferred choice when insertion at arbitrary positions occur frequently.

Nevertheless, it is possible that the implementation of all members of a program family utilizes the same resources. This can be observed in SGI's implementation[98] of STL where the four associative container classes `set`, `map`, `multiset`, and `multimap` are implemented by a single red-black tree[25] template class.

2.3 Domain Engineering

When software systems are developed within the context of a certain *domain* then the common requirements usually induce common characteristics in the systems. *Domain engineering* is a systematic approach to take advantage of the acquired knowledge and software artifacts obtained by developing systems within a domain. The goal is, using this experience and existing work products to deliver new products in a shorter time and at lower costs.

A *domain* is *an area of knowledge*[26] that includes not only a set of concepts and terminology understood by practitioners in that area but also knowledge of how to build software systems in that area. Examples of domains are database systems, numerical libraries, or graphical user interfaces.

This definition emphasizes how domain engineering involves knowledge both from the *problem* and *solution* domains.

There are two important points that are addressed by domain engineering. The first one is *engineering of reusable software*, e.g., libraries, frameworks or tools, for systems in a domain. Therefore domain engineering is sometimes traced back to Parnas' concept of program families [24, 26]. The second aspect is *knowledge management* which emphasizes that maintenance and update of knowledge in a domain is a continuous process that incorporates experience and new trends that are observed while developing the reusable work products.

2.3.1 Phases of Domain Engineering

According to Czarnecki and Eisenecker[26], domain engineering consists of three major phases or components, namely, *domain analysis*, *domain design*, and *domain implementation*. Figure 2.2 (which is based on the Figure in [26][p. 21]) gives an overview on the software development based on domain engineering.

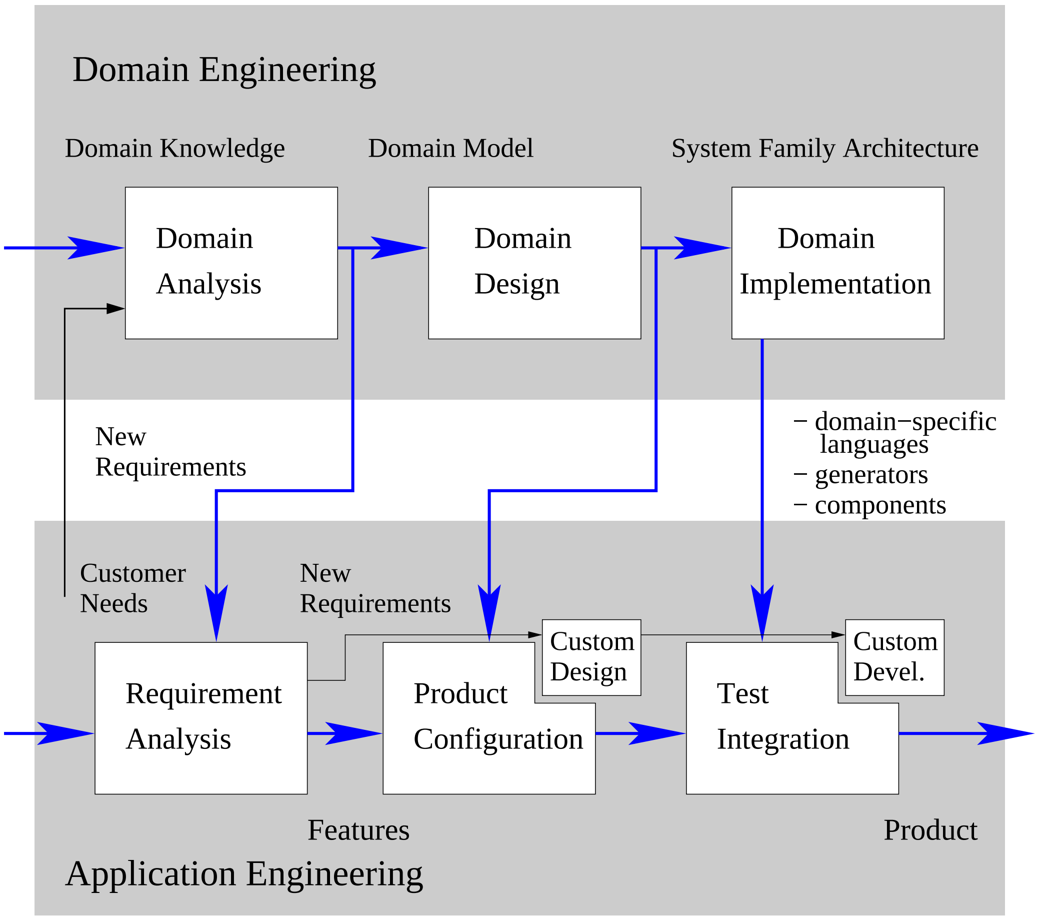

Figure 2.2: Software development based on Domain Engineering

Domain Analysis The objective of domain analysis is to choose and define the domain under consideration. Information about the domain is gathered and used to formulate general requirements for the systems in the domain.

Domain Design The objective of domain design is to develop a common architecture for the systems in the domain.

Domain Implementation In the phase of domain implementation, the reusable components, frameworks, tools, or libraries are implemented.

Figure 2.2 emphasizes how the products of domain engineering are reused during *application engineering*, i.e., when developing a particular system in a domain. It also indicates how the experience gained from building a particular system becomes part of the domain knowledge.

2.3.2 Domain Specific Languages

A particular approach to domain engineering is Weiss' *Family-oriented Abstraction, Specification and Translation* (FAST) method[107] that extends Parnas's work on program families.

Within FAST, commonality analysis is used to gather domain knowledge and to construct a structured vocabulary as a base for a *Domain Specific Language*[2] (DSL). The acquired knowledge on commonalities goes as a "design secret" into the DSL and becomes so part of all family members. The domain specific language keeps the "design secret" and is used to express variations between family members.

Domain specific languages can simplify the task of *rapid prototyping.* However, they also have disadvantages. Coplien[24][p. 20] mentions the cost of initial construction and of long-term maintenance of the tools and support staff. If these costs can't be spread across several projects then developing a DSL can be too expensive.

2.3.3 PROMOTER as a Domain Specific Language

The PROMOTER[45] parallel programming language was developed at GMD-FIRST as part of a programming environment for data-parallel scientific applications. In PROMOTER which is realized as a parallel language extension[32] of C++, the programmer specifies problem-specific[39, 53] data topologies and communication topologies for an abstract parallel machine.

[2]Weiss uses the term *Application Modeling Language* (AML).

Here the domain specific language is realized as a parallel language extension of a popular programming language. The design secret is the task of mapping the graph structure of a parallel application onto the underlying parallel machine and performing coordinated operations on distributed data.

Although domain engineering played no *explicit* role in the PROMOTER project the components of this design method have been applied implicitly.

Domain Analysis The chosen domain were parallel scientific applications as they occur as part of so-called *Grand Challenges* applications which include Global Climate Modeling, Quantum Chromodynamics, or Modeling Ultra-Low Loss Accelerators. The reusable work products of the PROMOTER project should relieve application programmers from the low-level aspects of parallel programming. These aspects include distribution of the huge data sets of these applications and the efficient handling of communication that occur while performing parallel operations on the distributed data. While this can be relatively easily achieved for regular applications, it is much more challenging for applications with irregular spatial patterns.

Domain Design The PROMOTER programming model introduces so-called *data topologies* and *communication topologies* as specifications of distributed data types. Jointly they describe the graph structure of arbitrary parallel applications. The task of mapping the problem oriented program graph onto a physical parallel machine is delegated to the compiler and run time system. The corresponding communication patterns between distributed processes, so error prone if done by hand, are automatically generated by the compiler or run time system.

Domain Implementation The reusable work product were the definition of the PROMOTER parallel programming language, a compiler that transforms PROMOTER programs to C++ program, thereby incorporating calls to Promoter Runtime Library[46] (PRL), and tools for partitioning large point sets[12].

Lessons Learned One of the lessons learned from the PROMOTER project was that creating (parallel) extensions to an already very complex language such as C++ creates many problems. As a result, it took a long time until a prototype of the PROMOTER compiler was available.

It was also very difficult to keep track with the evolution of C++ which was standardized during the time of the PROMOTER project. This reflects Coplien's statement on the construction and maintenance cost of domain specific languages in §2.3.2.

Another experience gained through PROMOTER was that although PROMOTER had constraints regarding the dynamic behavior[92] of the logical spatial structures of scientific applications, it neglected a clear separation of spatial structures and distributed data. This tremendously complicated the *efficient* implementation of irregular methods.

These deficiencies have been addressed in the successor project PROMISE[89]. Rather than defining language extension, the C++ template library Janus was designed and prototypically implemented by the author of this thesis.

2.4 Object-Oriented Design and Programming

Object-oriented design rests on the identification and construction of abstractions that are closely related with their real-world equivalents.

Objects represent concrete or conceptual entities and encapsulate their state and behavior. Classes describe objects with common attributes and operation implementations[3].

A programming language that provides linguistic support for objects is referred to as an *object-based* language. An object-based language is called *object-oriented* if the classes that describe objects can be *inherited*, that is, a more specific class (*subclass*) can acquire the attributes and operations of a more general class (*superclass*).

This widely accepted definition of object-orientation is due to Wegner[106] and can be concisely written as

$$\textit{object-oriented} = \textit{objects} + \textit{object classes} + \textit{class inheritance}.$$

Inheritance includes *polymorphism* of operations where a subclass can redefine the implementation but not the specification of an operation of a superclass. By searching the inheritance hierarchy the interface can be *dynamically* resolved to an implementation.

When object-oriented design methods evolved in the 1980-ies they were considered as a major advance over structural design methods and widely embraced in the industry. The main reason for this enthusiasm was that object-orientation rests on the encapsulation of state and behavior which can simplify the use, deployment, testing, maintenance and reuse of software units.

Also the scientific computing community appreciated these new ideas—as the following quote shows.

> Objects are natural metaphors for both physical objects and abstract entities. Expressing computations in terms of objects reduces the gap between concept and program.[9]

In particular inheritance was considered to be a way to express incremental modifications and extensions of existing classes.

The following example presents the object-oriented design of a simple matrix library. It highlights important drawbacks of the object-oriented design method when used in the field of high performance scientific computing. The example provides some reasons why object-orientation entered the field of scientific computing only partially.

2.4.1 An Object-Oriented Linear Algebra Library

Figure 2.3 shows a typical early example of object-oriented design of numerical libraries. The matrix library[50] is implemented in the relatively seldom used language Eiffel[74] but the crucial points of its design regard also C++[99] or Java[5].

The base class `MATRIX` declares all fundamental methods. There are *accessors* for reading and writing matrix elements and *operators* that perform high level computations like vector-matrix, matrix-matrix multiplications, or Cholesky decomposition.

The accessors are declared as *deferred*[3] methods and *must* be implemented by derived classes. For *operators*, default implementations are provided that *can* be redefined in the derived classes. The classes `LOCAL` and `DISTRIBUTED` represent basic deployment decisions for sequential and distributed programming environments. The library also provides several concrete distributed matrix classes—see Figure 2.3.

[3]A *deferred* methods in Eiffel corresponds to a *pure virtual* C++ function or an *abstract* Java method, that is, only a method signature is specified.

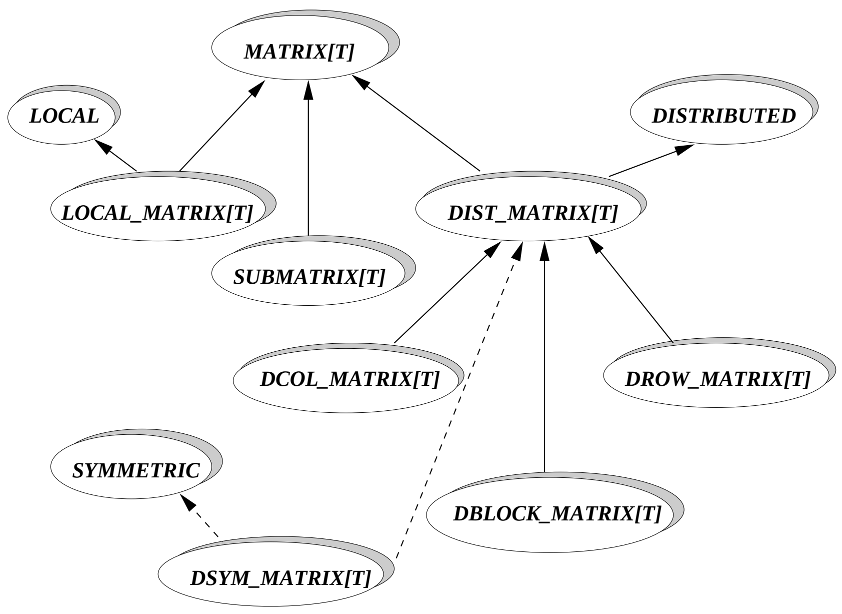

Figure 2.3: Example of a matrix class hierarchy

They claim that by deriving sequential and parallel matrix classes from the `MATRIX` base class, a user can easily replace sequential matrices through their parallel counterparts. Thus the parallelization would remain transparent to the user.

The drawback of this design is the high overhead related with calling the abstract accessor functions. In general, the selection of the actual method instance is deferred until runtime. The problem is not so much the additional level of indirection when calling a deferred method but that it often prevents optimizations that can be performed when the exact method instance is visible to the compiler. The most important of these optimization techniques is *function inlining* which not only completely removes the function call overhead but also enables cross function optimizations.

As mentioned above, the operator functions that implement higher level algorithms can be redefined in the derived classes to take advantage of additional information of the involved matrix types. However, since inlining the accessor functions is so crucial to obtain high performance this would require a substantial amount of custom design for each concrete matrix type and thus make the interface specified by the `MATRIX` class useless.

2.4.2 Subclassing and Subtyping

The use of inheritance when constructing object-oriented systems has three major aspects[100]:

- *Subclassing*, also called *implementation inheritance*,
- *Subtyping*, also called *interface inheritance*, and
- *Promise of substitutability*, which is also known as *Liskov Substitution Principle*[70].

The substitution principle simply says that functions that use references to base types must be able to use references to derived types without knowing the difference. The "two levels of programming" discussed in §2.4.1 depends on the substitution principle to hide the differences between sequential and parallel matrix classes.

Interface inheritance, that is the use of abstract methods, is closely related with *late binding* and, as discussed in §2.4.1, the performance penalties related with it are often considered unacceptable for the kernels of scientific software.

On the other hand, implementation inheritance as a technique for reuse, is often considered as bad design because it can break encapsulation. Moreover, it makes a derived class more *fragile*[100][p. 102] because its implementation becomes more likely to depend on internal details of its base class. Therefore, implementation inheritance is discouraged by many authors[9, 37] and *object composition* considered the better way.

One way to avoid explicit naming of a base interface is *structural subtyping.* In structural subtyping, a subtype is formed if a subset of the operations coincides with operations defined for another type[100][p. 78].

This is closely related to generic programming (see §2.6) where interfaces are implicitly formulated by syntactic and semantic *requirements.* The containers of the C++ standard library constitute a good example for structural subtyping. As mentioned in §2.2.2, all front insertion sequences of the STL are required to provide the methods `front`, `push_front`, and `pop_front`. There is however, no base class for STL containers that declares these methods. Rather each sequence class provides it own definition of these methods. Thus, different sequence types can be easily interchanged and no run time or space overhead occurs when using this implicit interface.

2.4.3 Toolkits

Object-oriented languages provide class constructs for encapsulating sets of definitions that are easily adapted for new programs[34]. Reusable classes often occur as part of *societies* that, as a whole, provide services that make them useful.

An example are the C++ standard library classes for input and output operations. This set of classes provides stream classes and predefined stream objects. There are also classes for stream buffering, stream formatting and manipulators—see Figure 2.4 that is based on the description given in [81]. In this example the associated classes are related through inheritance and common template parameters.

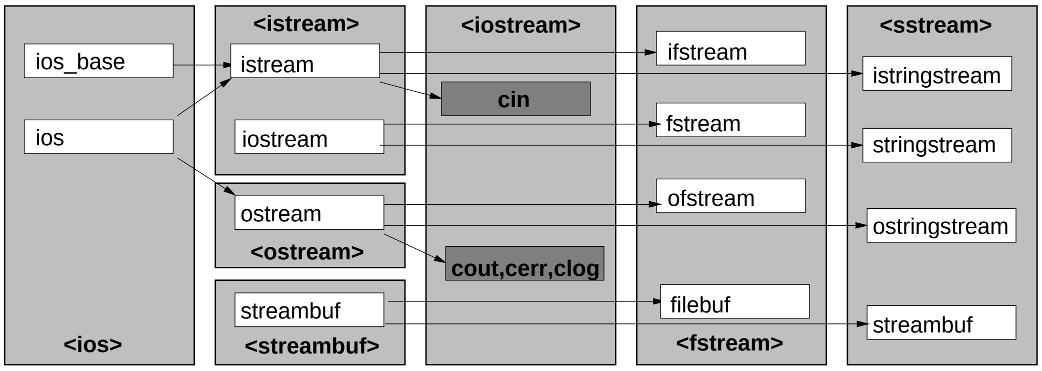

Figure 2.4: Main C++ stream classes and objects

Such societies of reusable classes are referred to as *toolkits*.

> A toolkit is set of related and reusable classes designed to provide useful, general-purpose functionality[37][p. 26].

The focus of toolkits is on *code reuse*.

2.4.3.1 Example: PETSc

The *Portable, Extensible Toolkit for Scientific Computation*[6, 7] (PETSc) was developed at Argonne National Laboratory and is primarily aimed for the numerical solution of partial differential equations and related problems on high-performance (parallel) computers.

PETSc is an *object-based* toolkit and implemented in C. The aim of PETSc is to create various solvers for linear and nonlinear systems of equations with a particular emphasis for managing the low-level infrastructure of parallel programming. PETSc has been used for finite element methods[2], optimization[14], computational fluid dynamics[1], wave propagation, and the Helmholtz equation[109]. Figure 2.5 gives an overview on the abstractions and services provided by PETSc.

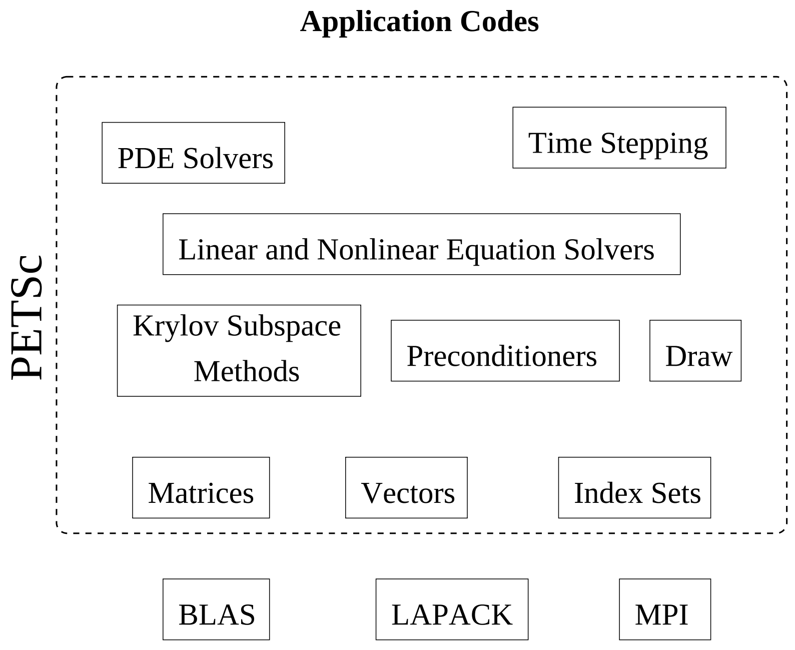

Figure 2.5: Overview on features offered by PETSc

PETSc represents an outstanding example of reuse in the field of scientific computing and can be considered as state of the art for parallel libraries in this field. It provide parallel vector and matrix types, scalable parallel preconditioners, Krylov subspace methods, Newton-based nonlinear solvers and more. It provides intensive error checking and is portably implemented on UNIX and Windows systems. Besides its direct use in scientific application programs, PETSc has been also interfaced to other scientific libraries[22]—for example Overture (see §2.4.4.1) and SAMRAI (see §2.4.5.1).

One of the reasons why PETSc was implemented in C and not in C++ was that at that time (1995) C++ was still on the way to be standardized. The definition of important language features, like C++ templates, were still in the flux. Moreover the availability of C++ on supercomputers was very restricted.

The restriction to C as implementation language is one of the reasons that PETSc is so portable. Moreover, it does not hinder its *external* usability. On the other hand, it does complicate the *internal reuse.* For example, the implementation of ordinary and block-structured matrix vector operations are to a large extent unrelated[96]. Also the set of matrix and vector element types is mostly restricted to simple builtin types.

2.4.4 Frameworks

When object-oriented techniques were adopted to scientific computing, developers often concentrated on the efficient implementation of concrete classes (or simple class hierarchies) to represent vector or matrices. Later it was observed that object-oriented design can offer benefit also on higher levels of scientific software. The basic idea is to create *frameworks* that decompose complex algorithm into smaller parts that can be tested and maintained more easily. Moreover, frameworks also promote code and application reuse.

We use the following definition of the term *framework.*

> A framework is a set of cooperating classes that make up a reusable design for a specific class of software[37][p. 26].

A framework dictates the architecture of an application since it defines its overall structure, the partitioning into classes, and their key responsibilities. In contrast to earlier object-oriented reuse techniques based on class libraries, frameworks are targeted for particular application domains and emphasize *design reuse.*

In the field of parallel scientific computing the most notable frameworks are Overture for applications on block-structured grids (see §2.4.4.1), SAMRAI for structured adaptive mesh refinement (SAMR) (see also §2.4.5.1), and POOMA[56, 61, 62] for particle simulation. The next subsection gives an overview on the Overture framework.

2.4.4.1 Example: Overture

Overture[21, 20] is an object-oriented code framework for writing parallel structured adaptive mesh refinement methods (SAMR). It is implemented as a collection of the parallel C++ libraries that enable the use of finite difference and finite volume methods and hides the details of the parallel implementation. The framework includes among others the A++/P++ array class libraries to represent grids and grid functions, AMR++ which provides abstractions for adaptive mesh refinement (AMR), and support for load balancing for a collection of structure grids. Figure 2.6 shows the main parts and layers of the Overture framework.

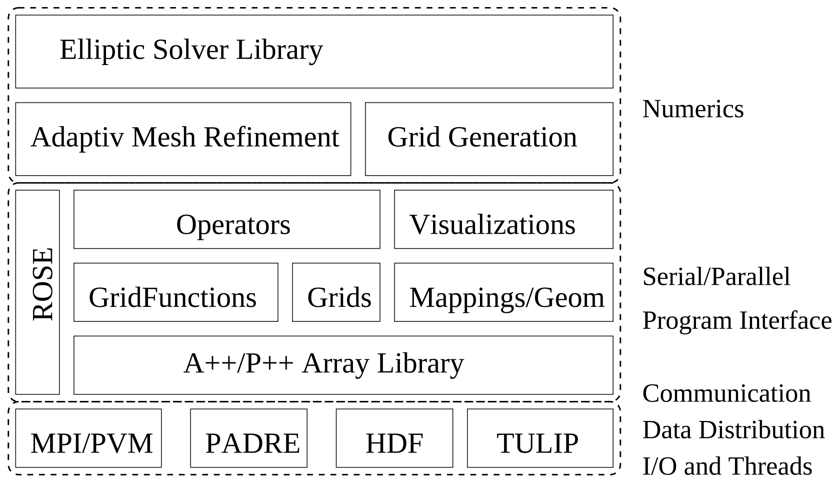

Figure 2.6: Overview on the Overture framework

Overture is designed for solving problems on a structured grid or a collection of structured grids and can *not* be used for finite element simulations on general meshes.

Figure 2.6 indicates that the Overture framework includes not only discretized differential operators but also solvers for elliptic partial differential equations. On the other hand, it has been pointed out[22] that although the Overture data structures are well optimized for the discretization of partial differential equations they are not optimal for linear algebra operations which are at the heart of solver package. As mentioned in §2.4.3.1, Overture can utilize PETSc for the efficient solution of the linear systems. For converting Overture data structures to PETSc data structures a peer to peer approach has been chosen[22].

2.4.5 Design Patterns

A *design pattern* names, abstracts, and identifies the key aspects of a common design structure. It accomplishes this by identifying the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. The pattern catalog presented in *Design Patterns*[37] is widely accepted.

Design patterns are often used in building object-oriented systems but are by no means restricted to them. For example, the *Boost Graph Library* (BGL)[67] rests on the paradigm of generic programming §2.6 and is *not* an object-oriented library. BGL uses the *Visitor* pattern[37] to extend and customize the various steps of graph algorithms.

The book of DṠchmidt, et. al., on patterns for concurrent and networked objects[90] must also be mentioned in this context. The patterns presented there deal with event handling, concurrency, and synchronization. They are very useful for client-server applications but not for data parallel applications.

2.4.5.1 Example: SAMRAI

SAMRAI[55, 65] stands for *Structured Adaptive Mesh Refinement Applications Infrastructure* and provides computational scientists with general and extensible software support for rapid prototyping and development of parallel *structured* adaptive mesh refinement (AMR) applications. The attribute *structured* means here (as in the Overture framework presented in 2.4.4.1) that meshes are considered as collections of regular grids (called *patches*). Thus, in accordance with Overture §2.4.4.1 and POOMA[62], it cannot be used for general finite element simulations.

Structured adaptive mesh refinement methods require a complex software infrastructure and it is highly desirable that its design is reasonably flexible so that the infrastructure can be reused for a broad range of applications.

SAMRAI uses several standard design patterns[37] to provide a flexible framework design[55]. Among these patterns are

Smart Pointer which is used for type safe dynamic casting and memory management of shared objects.

Abstract Factory that is an approach for creating families of related objects without specifying their concrete types. This pattern is used to integrate user defined data types into SAMRAI.

Strategy to define and encapsulate families of algorithmic components. In SAMRAI is is used to define a family of time stepping algorithms, see Figure 2.7.

Figure 2.7 gives an example of *Strategy* pattern. The intent of the *Strategy* pattern is to define a family of algorithms that are interchange-

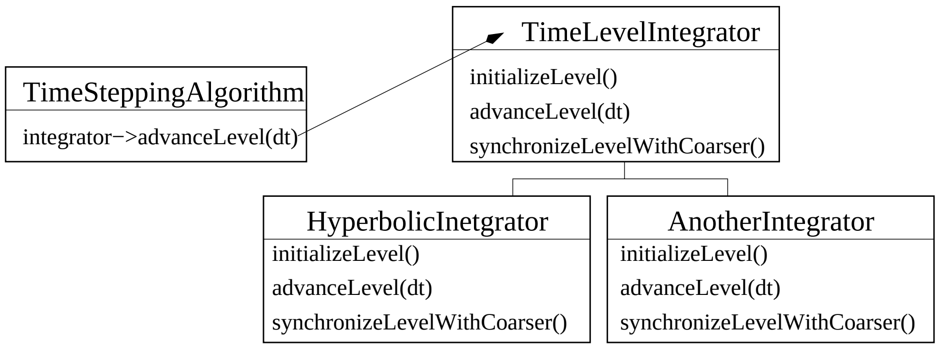

Figure 2.7: The Strategy pattern used for time stepping algorithms

able. This pattern provides an alternative to inheritance which expresses a very close coupling of classes. Algorithms implemented with the Strategy pattern can be dynamically exchanged. In this example, the abstract base class `TimeLevelIntegrator` declares the methods `initializeLevel` or "`advanceLevel`. Integrator classes that are derived from this abstract class provide concrete time level integration procedures. The client, here represented by the class `TimeSteppingAlgorithm`, only refers to the abstract time level integrator. Thus, the application can exchange the algorithms at run time. The overhead related with this extra indirection is negligible for complex algorithms. A potential drawback is that the application typically must be aware of the semantics of the different concrete strategies.

2.5 Aspect-Oriented Programming

The *principle of separation of concerns*[29] is a fundamental engineering principle approach to create quality software. A problem however is, that often important design decisions cannot be clearly encapsulated into individual *functional units*. Rather there are often issues that *cross cut* the basic functionality of a system.

One example for this is *error handling* in a software system where exceptional situations that occur in one class (or module or procedure) is often treated in other subsystems. Error handling is often expressed by small code fragments that are scattered over several subsystems. Other examples are security control, synchronization, or memory access patterns.

The paradigm of *aspect-oriented programming* (AOP)[64] explicitly addresses the problem of design and implementation *aspects* that cross cut several *components*. It is a programming technique that aims at an appropriate isolation, composition and reuse of aspects.

Kiczales et al.[64] understand by a *component* a property that can be cleanly encapsulated in a *generalized procedure* (i.e. class, module, procedure). Cleanly means, well-localized, easily accessible, and composable. An *aspect* on the other hand, cannot be cleanly encapsulated in a generalized procedure.

Interesting for the development of software for scientific computing is the following remark of Kiczales[64].

> Many performance-related issues are aspects, because performance optimizations often exploit information about the execution context that spans components.

From this point of view, *data distribution* is an aspect of a scientific simulation because it cross cuts several (sometimes all) components of a parallel simulation.

2.5.1 Structure of Aspect-Oriented Programming Systems

Aspect-oriented programming is still in a definition phase[26][p. 253]. Kiczales has proposed the following structure of a AOP system.

1. (a) a *component language* for the functional units (components),
   (b) one or more *aspect languages* in order to program the aspect,
2. an *aspect weaver* for the combined languages,
3. (a) a *component program* implemented in the component language,
   (b) one or more *aspect programs* which implement the various aspects.

In analogy to *method binding* in object-oriented languages, aspect weaving can be performed at compile time or be delayed until runtime.

2.5.2 Example: Aspect-Oriented Sparse LU Factorization

An aspect-oriented programming environment for matrix computations has been presented in[57]. This environment allows a user to write sparse matrix code on a high level along with annotations that allow an efficient implementation. It is pointed out that efficient sparse matrix code requires a careful selection of data structures. In particular, it must be avoided to perform operation on zeros of a sparse matrix. *Operator fusion* is also an important issue to achieve high performance. These aspects are usually explicitly dealt within lower lever C or Fortran libraries which result in less readable programs that are more difficult to maintain than code written in MATLAB.

The proposed environment has a component language that is similar to MATLAB[23]. Aspects that have to be considered, are data representation aspects (e.g. matrix formats and orientations) and permutation of vectors or matrices. These aspects are represented by special annotations, e.g., declaration of the matrix structure and a *view* annotation that allows to view a matrix or vector through a permutation vector. The aspect weaver parses the annotated program and generates C++ code. The run time of the generated code is comparable with a standard Fortran library[57].

The paradigm of aspect-orientation is a very promising approach to deal with the problem of different members of a program family. Individual members that are specialized for certain aspects can be *generated* from an annotated program using an aspect weaver. On the other hand, an aspect weaver and the related aspect languages can result in a complex software infrastructure that poses the question of costs and maintenance (see also §2.3.2).

2.6 Generic Programming

The paradigm of *generic programming*[79] aims at simplifying the development of libraries where a family of algorithms have to be implemented for many data structures. Generic programming is *programming in terms of concepts*[77], in contrast to programming in terms of individual data structures. The term *concept* is defined as a family of abstractions that are formed by a common set of requirements. Musser[78] points out that

> A large part of the activity of generic programming, particularly in the design of generic software components, consists of concept development—identifying sets of requirements that are gen-

eral enough to be met by a large family of abstractions but still restrictive enough that programs can be written that work efficiently with all members of the family.

Generic programming is obviously related with program families (see 2.2). However, program families present a much broader approach that is neither restricted to the problem of simplifying algorithm development for a set of data structures nor bound to particular implementation strategies. Generic programming relies on special language support such as the presence of generic types or generic functions. The described language support emphasizes that the interoperability of algorithms and data-structures is not the only aspect of generic programming. Two other aspects are *element type parameterization* and *extension through function objects*.

In generic programming, a concept description consists not only of syntactic and semantics parts but also of *strict efficiency specifications*. As indicated in the above quote, efficiency is major concern because clients would tend to prefer a non-generic program over a much less efficient generic program. As a generic program can replace a large collection of non-generic programs, it has the great advantage of simplified maintenance.

Generic programming[79] has become widely known since it was used to design Standard template Library (STL), that is, the algorithm and container framework of the C++ standard library[98, 99]. The essential points of this framework are discussed in the following subsection §2.6.1.

2.6.1 The Standard Template Library

The containers of the Standard Template Library (STL) together with their conceptual hierarchy have been introduced in §2.2.2. The point of STL, however, is not so much the template container classes but rather how STL algorithms are described so that they can work on the containers in a data-structure neutral way.

For example, STL provides the template function `find` that implements a linear search algorithm. The following code shows a prototypical implementation of STL's `find` method[98].

```
template <typename InputIterator, typename EqualityComparable>
InputIterator find(InputIterator first, InputIterator last,
                   const EqualityComparable& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

The keyword `template` indicates that the function `find` defines in fact a *family of functions* for the argument types defined in the template parameter list. The first two arguments `first` and `last` describe a range in which the third argument `value` is searched. A range is given by the *right open* interval `[first, last)`.

To apply this algorithms to any container type `Cont` instantiated by one of the seven container templates of Figure 2.1 one would supply the iterators returned by the `begin` and `end` member functions.

```
Cont c;
Cont::value_type v;
//initialize c and v
Cont::iterator it = find(c.begin(), c.end(), v);
```

This is possible since all STL containers provide nested iterator types that can be

- *tested for equality* with the operators `==` or `!=`,
- *dereferenced* using operator `*`, and
- *incremented* using operator `++`.

This example shows how STL algorithms are decoupled from the actual containers. The algorithms do not directly work on the containers but rather on *iterators* as shown on Figure 2.8.

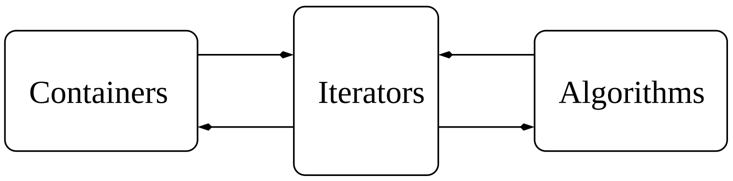

Figure 2.8: Decoupling of containers and algorithms through iterators

Any type that fulfills the syntactic and semantic requirements that the STL imposes on the first template parameter of `find` can be used to call `find`. The precise set of requirements are given by the STL concept *input* *iterator*. STL provides a whole hierarchy of *iterator concepts* that are all broad enough so that their requirements are satisfied by ordinary C pointers. Therefore the `find` algorithm can also applied to search a value in a C array as the following code fragment shows.

```
int a[100000];
// ...
int v = 12;
int* p = find(a, a+100000, v);
```

The interoperability with a wide range of data types is a key advantage of the STL. Object-oriented design techniques, on the other hand, often rely on a common base class for types that can be used within their container framework. For example, the Java *Collections* framework[5] cannot directly deal with built-in types such as `int` because they are not derived from `Object` class. Moreover, the use of STL algorithms on pointers as iterators causes no runtime overhead when compared to the following non-generic implementation of linear search.

```
int* find_int(int* first, int* last, const int& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

A decent C++ compilers generates for the function `find_int` object code that is as efficient as that for the function `find<int*>`.

2.6.2 Generic Matrix Libraries

The task of simplifying algorithm development for various data structures is particularly pressing for sparse matrix computations. Sparse matrices play a central role in scientific software, in particular, when solving linear systems that result from the discretization of partial differential equations. As mentioned in the introduction of this chapter, there are more than forty sparse matrix formats in use[73].

A generic programming approach could avoid the problem of code explosion because it would not be necessary to have for each sparse matrix data structure its own sets of algorithm. Beside this, a generic library could be easily parameterized over a broad range of element value type. This would cover not only predefined types such `int`, `double`, or `complex<float>` but also user defined types. Conventional libraries, for example PETSc which was introduced in §2.4.3.1 can only be used for a small set of types that are known at library development time.

The Matrix Template Library MTL[72, 95] is a generic library that provides comprehensive linear algebra functionality for a wide range of matrix formats including sparse ones. As the Standard Template Library (STL), MTL has a conceptual framework of containers, iterators, adaptors, and function objects. To traverse and access different matrix data structures so-called *two-dimensional iterators* are introduced.

MTL comprises many generic matrix algorithms including matrix-vector multiplication, matrix-matrix multiplication and routines for the solution of triangular systems. Closely related to MTL is the Iterative Matrix Library[71] (ITL) that provides iterative methods for solving linear systems. ITL also includes various preconditioners. ITL uses the interface of matrix-vector, vector-vector, and vector-scalar operations specified by MTL. However, ITL can also use other packages such as Blitz++[105].

To achieve high performance MTL relies on static polymorphism, automatic loop unrolling, instruction scheduling and algorithmic blocking. This brings MTL close to *generative* programming techniques that are discussed in §2.7. A drawback of MTL is that it does not directly address the problem of parallel and distributed matrices.

A new variant of the generic programming paradigm has been presented by Mateev, Pingali, and Stodghill[73] to deal more directly with the above mentioned code explosion problem for sparse matrices. They propose *two* generic APIs. The first API allows designers to express generic matrix algorithms in an array notation. The second API exposes the details of (compressed) sparse matrices. Restructuring compiler technology[13] is used to transform one API into the other.

2.6.3 Generic Graph Libraries

The paradigm of generic programming has also been creatively applied to the domain of graph algorithms. An outstanding example is the Boost Graph Library[67] (BGL) that is part of the Boost[28] library collection of free peer-reviewed portable C++ source libraries. The emphasis of this collection is that they work well Standard Library of C++. BGL was formerly known as Generic Graph Component Library[68, 69].

BGL provides an interface to abstract the details of particular graph data-structures. BGL introduces its own iterators to define different graph traversal patterns. Similar to element type parameterization of STL, BGL graphs can be customized with user-defined *properties*.

Generic algorithms of BGL rest on a small set of algorithm patterns which themselves are implemented as generic algorithms. The basic algorithm patterns are breadth first search, depth first search, and uniform cost search. The graph algorithms in BGL include among others Dijkstra's and Bellman-Ford Shortest Paths, Reverse Cuthill Mckee Ordering, and Topological Sorts.

The algorithms can be extended function objects with multiple methods that are called at several "event points" of an algorithm. These functions objects are referred to as *visitors* and are closely related to the *Visitor* design pattern[37]. This customization of algorithms by visitor objects is also applied in the Janus template library—see Chapter 4.

Similarly to MTL, the Boost Graph Library can only be used for non-distributed graphs. This restriction is addressed by the Janus template library that provides data structures and algorithms that can efficiently both on distributed memory and shared memory architectures.

2.7 Generative Programming

Czarnecki and Eisenecker give the following definition of *generative programming*:

> Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirement specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge[26].

Generative programming is aimed at program families §2.2 and represents a particular way of domain engineering §2.3.

Generative programming requires a *means of specifying* family members, *implementation components* as a generation base, and *configuration knowledge* that maps from a specification to a product.

Traditionally, generative programming relies on *code generators* that are outside the used programming language—see §2.7.1. In C++ however, for some years *template metaprogramming* has used to generate tailored configurations of classes and functions. Whereas traditional object-oriented design and programming techniques rely on late binding, template metaprogramming uses the compiler itself to run code that determines what program is produced.

An example for the latter approach is the Generative Matrix Computation Library[26] (GMCL). This library uses template meta programming for the design, configuration, and generation. It is comparable with the Matrix Template Library[72] that has been presented in section 2.6.

2.7.1 The AEOS Way

AEOS stands for Automated Empirical Optimization of Software and is a new paradigm for the production of highly efficient routines on modern high performance computing platforms[108]. It arose from the observation that the efficient implementation of standard numerical software (e.g. linear algebra kernels) hardly keeps pace with the hardware development. Providing an optimized implementation of BLAS[31] (Basis Linear Algebra Subroutines) for a new processor, for example, requires several man-month of highly trained (both in linear algebra and computational optimizations) personnel.

For supporting a library with the AEOS methodology several requirements must be fulfilled.

- Isolation of performance critical routines.
- A method of adapting software to different computing environments. This may include parameterized code or configurable code generators.
- Robust, context-sensitive timers which is important to produce reliable input for a generator.
- Appropriate search heuristics. This is necessary because an advanced code generator can have hundreds of ways to implement an operation. Therefore the search trees must be rapidly (yet reliably) pruned.

2.7.2 Application to Linear Algebra Kernels

The AEOS methodology has been successfully applied to the BLAS[31] library. ATLAS[108] which stands for Automatically Tuned Linear Algebra

Software, uses code generators in order to produce many different ways a given linear algebra operation can be performed. It concentrates on the Level 2 BLAS (matrix-vector) operations and Level 3 (matrix-matrix) operations because here optimizations can achieve remarkable result. This is because the loop structure of a conventional BLAS implementation is often too complex for a compiler to figure out appropriate configurations.

ATLAS relies on an adequate C Compiler because the code generator produces ANSI-C code. There is however a tension between the code transformations and requests for registers performed by ATLAS on the one hand, and optimizations usually performed by a C compiler on the other hand. Sometimes compilers ignore register requests[4] that are present in the source code. In the context of ATLAS this can yield poor results.

ATLAS code generation strategies include register blocking, loop unrolling, and choice of floating point instruction. It also relies on the presence of *hierarchical memory*, at least, registers and a L1 data cache. Otherwise, ATLAS blocking and register usage can turn into overheads[108].

Experiments[108] show that the speed of the ANSI-C code generated by ATLAS is comparable with vendor-tuned implementations and often an order of magnitude faster than a reference implementation of BLAS. Installing ATLAS and generating the code usually requires only an hour. This compares very favorably with the many man-months of hand-tuning mentioned above.

2.8 Component-Oriented Programming

Component-oriented development of software is considered as an evolutionary step beyond object-orientation[100]. Component programming means an encapsulation of units of functionality and providing a (usually meta-language) specification of their interfaces[4].

It arose from the observation that object-oriented techniques did not result in significant amount of cross-project code reuse. This is related to incompatibilities of object-oriented languages and their need for compile-time binding of interfaces.

Component-orientation focuses on issues of language independence and defines standards for communication among (distributed) components. A short definition of the term component is from Szyperski[100]:

[4]The C language family has the keyword `register` to *hint* register usage to a compiler.

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Several technologies have evolved within the software industry to support component programming, namely, CORBA[75], COM[49], and JavaBeans[76].

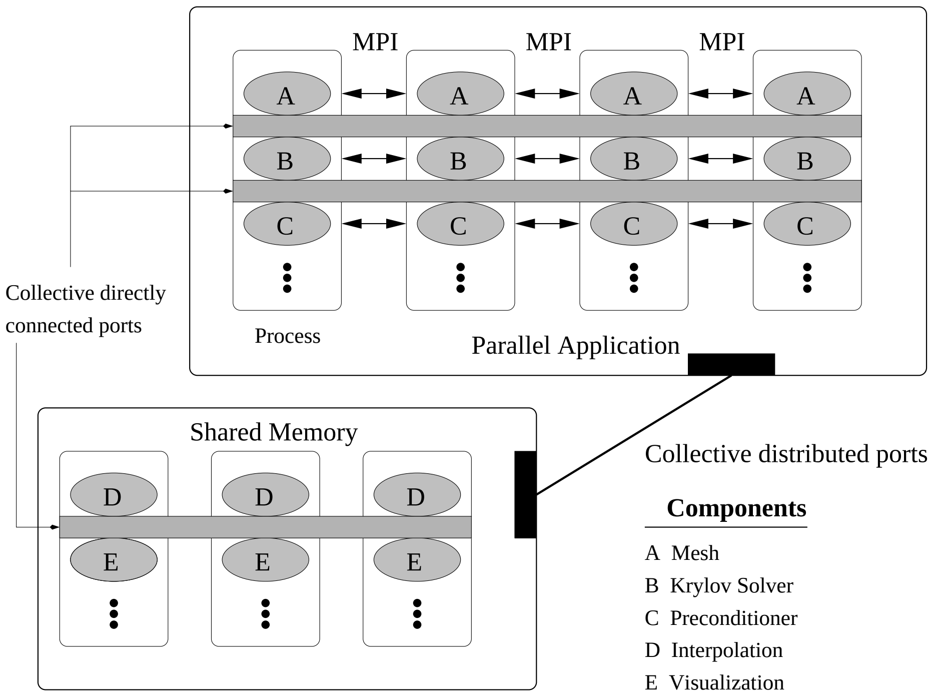

Figure 2.9: Component interactions in parallel applications Tightly coupled numerical components, such a Krylov solver (B) and a Preconditioner (C), require connections with high bandwidth and low latency. These components however, can be parallel programs that uses MPI for intra-component communication. The CCA proposes so-called *collective directly connected ports* to handle interactions among parallel components. Visualization components can often be more loosely coupled to the numerical components. For connecting such components *collective distributed ports* are used.

In the field of high performance scientific computing, component technology is very promising since the applications are becoming more and more complex and require multidisciplinary development teams. The challenge are the constraints to achieve *high performance*. This is why the industry standards are currently considered as not usable for tightly coupled parallel scientific applications[4].

The goal of the Common Component Architecture[38, 4] (CCA) is to develop software component technology for high-performance parallel simulation software. The main motivation is—as mentioned above—that component approaches based on CORBA[75], COM[49], and JavaBeans[76] technologies do not address parallelism. The different needs of components in parallel applications are sketched in Figure 2.9.

The CCA modifies these traditional approaches to suit the needs of high-performance computing and to bring this advanced software technology to the scientific community. Established in 1998, the CCA is still in a definition phase[5]. For this reason the PAWS (Parallel Application WorkSpace) approach to components to parallel components is considered here.

2.8.1 Parallel Application WorkSpace

The Parallel Application WorkSpace (PAWS)[10, 11] was developed at the Los Alamos National Laboratory. It proposes a software infrastructure to connect different parallel applications within a component-like model.

A central PAWS Controller (similar to an object request broker) coordinates the connection of sequential or parallel applications across a network. PAWS addresses various challenges[11]

- dynamic coupling of applications during their execution,
- avoid serialization bottlenecks during data transfers between parallel applications,
- sharing of parallel data structures (in the presence of different layout strategies),
- support for data exchange between applications written in different programming languages.

Which data structures are to be shared and at what points the data are exchanged, must be specified with the PAWS API which is implemented as a C++ class library. PAWS provides a general parallel data descriptor, and

[5]As of the time writing (March 2001) the most recent proposal is from January 2001

automatically carries out parallel layout remapping when necessary. It does not provide an *interface definition language* (IDL). In general, PAWS has a data-centric view and does not focus on computational steering. A major limitation of PAWS is that is restricted to rectangular data arrays which excludes many irregular applications.

PAWS provides APIs for C++, C, and Fortran. Connections between distributed components can be dynamically established and canceled. PAWS uses the Nexus[86] communication library of the Globus[87] toolkit that provides software tools to simplify building *computational grids*[36]. Nexus is independent of the application's parallel communication mechanism.

2.9 Conclusions

In this chapter a wide range of software design techniques has been considered for their applicability in the field of scientific applications. Since efficiency is an outstanding concern when judging the quality of scientific software the question was how other quality aspects such as maintainability and portability can be increased without sacrifying performance.

In particular *generic programming* combined with design patterns present a promising approach to design and implement efficient, portable and extensible kernels of scientific codes. Component frameworks can be used to combine individual building blocks to larger multi-disciplinary applications.

The C++ standard template library STL[98], the Boost Graph Library BGL[67], and the Matrix Template Library MTL[72] emphasize the expressiveness and efficiency of components engineered using generic programming techniques. However, all three libraries do not address the issues of parallel and, in particular, distributed programming platforms. These challenges are addressed by the Janus conceptual framework and template library that will be presented in the Chapters 4 and 5. However, before abstractions for data parallel applications can be devised a proper analysis of the requirements of this domains must be performed. This domain analysis will be undertaken in the next chapter.

One obvious advantage of the generic programming paradigm and C++ templates is that generic components can be easily parameterized with user-defined data types. Non-generic frameworks and toolkits, such as PETSc[6], often suffer from a complicated customization process.

The broad range of discussed design paradigms makes it obvious that there is *no single, all-encompassing* design methodology. Therefore is has become popular to speak of *multi-paradigm design*[24]. This favors C++ as an implementation language[99] because it supports data-abstraction, object-orientation, and generic programming without abandoning the efficiency of the C programming language.

Chapter 3

Domain Analysis for Data Parallel Scientific Applications

A *domain analysis* starts (as described in §2.3) with a definition of the domain under consideration. In this thesis "data parallel applications" are those that that mainly rely on *data parallel algorithms* which are defined as algorithms whose

> parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control[54].

In §3.1 several important examples of data parallel applications are analyzed. Special emphasis is put on gathering information about data parallel algorithms that involve *irregular* data sets. In §3.2 the gathered domain knowledge is used to formulate general requirements for data parallel applications on regular and irregular data sets.

3.1 Examples of Data Parallel Algorithms

This section presents several examples of data parallel applications. This selection encompasses both applications that rely on *regular* data sets and those that involve *irregular* sets.

The class of regular applications includes a simple finite difference method §3.1.1 and the *Game of Life* cellular automata simulation §3.1.2. Examples for applications on irregular or dynamically created data sets are the Bellman-Ford graph algorithm §3.1.5, finite element discretizations §3.1.3, and several mesh refinement procedures §3.1.4.

3.1.1 A Simple Finite Difference Method

A simple boundary value problems is the Poisson equation on the unit square $\Omega = (0,1) \times (0,1)$. When imposing homogeneous Dirichlet conditions this partial differential equation reads

$$-\frac{\partial^2 u}{\partial x^2}(x,y) - \frac{\partial^2 u}{\partial y^2}(x,y) = f(x,y) \qquad (x,y) \in \Omega \tag{3.1}$$

$$u(x,y) = 0 \qquad (x,y) \in \partial\Omega. \tag{3.2}$$

The basic idea of the *finite difference method*[19, 52] is to find an approximate solution of a boundary value problem for a finite subset G of $\overline{\Omega}$ by replacing partial derivatives through difference quotients. The differential equation turns then into a (finite) system of algebraic equations for the approximate values in G.

Finite difference methods often uses *rectangular grids* as the finite subset of the domain. For the example at hand one can deploy the square grid

$$G = [0,N] \times [0,N] \tag{3.3}$$

as shown in figure 3.1. The number $h = \frac{1}{N}$ is called the *mesh width* of this equidistant grid. In this figure, boundary grid points are represented by $\bullet$ and inner grid points by $\circ$. The value of a function v in a grid point (ih, jh) where $(i,j) \in G$ is denoted by v_{ij}.

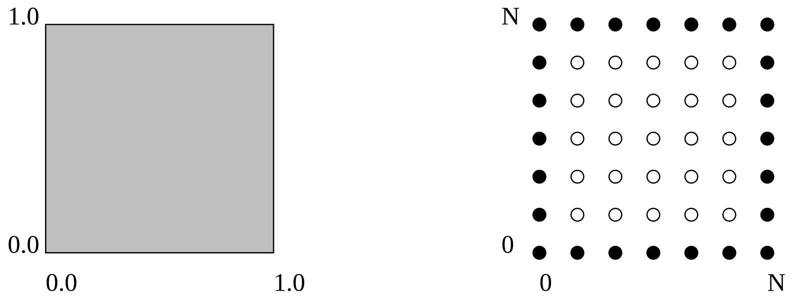

Figure 3.1: A two-dimensional grid for the unit square

A simple approximation of the second derivatives in the left hand side of Equation 3.1 for the inner grid points can be obtained by using a so-called *five-point finite difference stencil*[52]. This means that for $0 < i, j < N$ the following approximation holds

$$\left(-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2}\right)_{ij} \approx \frac{1}{h^2}\left(4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}\right).$$

Thus the discretization of Equation 3.1 reads

$$\frac{1}{h^2}\big(4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}\big) = f_{ij} \quad 1 < i,j < N-1. \tag{3.4}$$

Note that these equations relate each inner grid point (i,j) with five neighbor points

$$(i,j) \rightarrow \left\{ \begin{array}{ccc} & (i,j+1) & \\ (i-1,j) & (i,j) & (i+1,j) \\ & (i,j-1) & \end{array} \right\}. \tag{3.5}$$

Hence the name *five-point stencil.*

The simplest way to compute an approximate solution of this system of equations is to use the *Jacobi* iterative method[52]. For the system of Equations 3.4, a Jacobi step reads

$$u_{ij}^{n+1} \leftarrow \frac{1}{4}\left(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n + h^2 f_{ij}\right) \tag{3.6}$$

for all inner grid point (i,j).

Note that the computation of u^{n+1} for a certain grid point is independent from that in other grid points. This operation can therefore, within one iteration, be performed *in parallel* for all grid points.

3.1.2 Conway's Game of Life

Conway's Game of Life is a well-known cellular automata simulation that involves rectangular grids and a simple stencils to express data dependences.

In *Life*, the evolution of a population of cells is considered. The underlying spatial structure is a rectangular grid (see Figure 3.2). A value of 1 represents a living cell (marked by •), a value of 0 represents unoccupied grid points (not marked). The cells change their state depending on the number of neighboring living cells and their own state.

The neighbors of a grid point are defined by the eight-point stencil

$$(i,j) \rightarrow \begin{pmatrix} (i-1,j+1) & (i,j+1) & (i+1,j+1) \\ (i-1,j) & & (i+1,j) \\ (i-1,j-1) & (i,j-1) & (i+1,j-1) \end{pmatrix}. \tag{3.7}$$

Note that in this relation a grid point (i,j) is *not* a neighbor of itself.

If G is a two-dimensional rectangular grid and

$$a : G \longrightarrow \{0,1\} \tag{3.8}$$

a grid function that represents a population on G then the major part of the determination of the next generation of cells consists of using the relation in Equation 3.7 to compute the following sum

$$s_{ij} = \begin{cases} a_{i-1,j+1} & + & a_{i,j+1} & + & a_{i+1,j+1} & + \\ a_{i-1,j} & + & & & a_{i+1,j} & + \\ a_{i-1,j-1} & + & a_{i,j-1} & + & a_{i+1,j-1} & \end{cases} \tag{3.9}$$

for all $(i,j) \in G$. As in the case of the Jacobi iterative method of §3.1.1, the computation of this sum is completely parallel with respect to the grid points (i,j). Note that the sum s_{ij} is the *living neighbors* of grid point $(i,j) \in G$ with respect to the population $a : G \rightarrow \{0,1\}$.

With these notations the exact rules for the determination of a new generation

$$b : G \longrightarrow \{0,1\}$$

at an arbitrary grid point (i,j) read:

$$b_{ij} \leftarrow \begin{cases} 1, & \text{if } a_{i,j} = 1 \text{ and } s_{ij} \in \{2,3\}, \\ 1, & \text{if } a_{i,j} = 0 \text{ and } s_{ij} = 2, \\ 0, & \text{otherwise.} \end{cases} \tag{3.10}$$

Figure 3.2 shows the application of the rules in Equation 3.10 to a particular population.

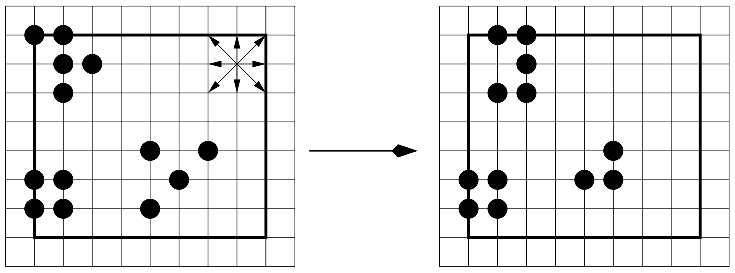
Figure 3.2: Two successive generation in *Life*

3.1.3 Finite Element Methods

The finite sets and their relations that occur in scientific applications can only in simple cases be described by rectangular grids. This holds in particular when complex geometries or local characteristics have to be described.

The method of *finite elements* is a family of approximation methods for partial differential equations that in contrast to finite difference methods can easily handle general geometries and non-continuous coefficients. Rather than replacing partial derivatives by difference quotients, a variational formulation of the boundary problem that constitutes an equation in an infinite-dimensional function space is approximated by a family of problems in finite-dimensional function spaces.

3.1.3.1 Basic Concepts

The basic idea of finite elements is to subdivide an N-dimensional domain Ω into a finite number of elements with simple geometry. The functions on Ω are approximated by functions that have a simple definition on the elements—often they are polynomials of a fixed degree.

Finite elements are distinguished by their shape and the degree of the approximating polynomials. For two-dimensional problems, triangles and rectangles are often used element shapes. In three dimensions hexahedra, tetrahedra, or pyramids are common choices—see Figure 3.3.

The approximating functions are usually polynomials on the elements and thus uniquely determined by the values, derivatives, or averages that can be related with certain points of the elements. These points are called the *nodes* of the finite elements. Figure 3.4 shows the nodes for linear, quadratic, and cubic polynomials on a triangle element.

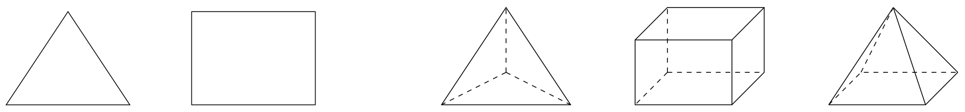
Figure 3.3: Shapes of common finite elements

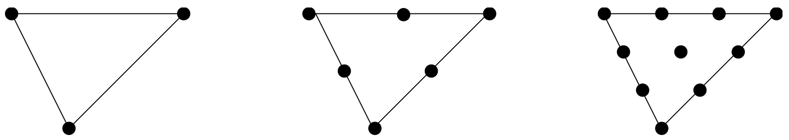
Figure 3.4: Nodes for different class of polynomials.

This means that a finite element program has to deal not only with the sets of elements but also with the involved nodes that are shown in the right part of Figure 3.5. The node set is the domain of definition of the approximating functions. Figure 3.5 shows the set of elements (i.e. a triangulation) and the corresponding node set for the case of linear triangle elements.

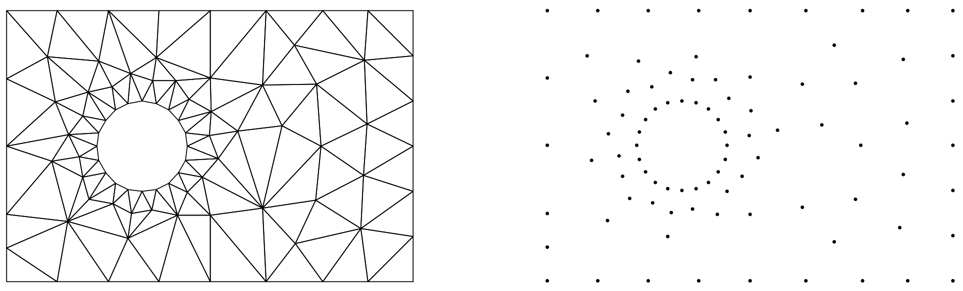
Figure 3.5: Elements and nodes of a finite element triangulation

3.1.3.2 Finite Sets and Their Relations

If I denotes the node set of a finite element discretization then the coefficients a_{ij} of the approximating system of equation usually form a sparse matrix. This means that the set

$$R := \left\{ (i,j) \in I \times I \,\middle|\, a_{ij} \neq 0 \right\}$$

has a cardinality that is $O(|I|)$. Note that the sparse matrix (a_{ij}) can be considered as a real (or complex valued) function on R

$$a : R \longrightarrow \mathbb{R}. \tag{3.11}$$

For large discretizations it is of utmost importance to exploit the sparsity of R when representing and solving the approximating system of equations. The sparsity pattern R reflects the connectivity of elements and nodes. If E denotes the elements of a finite element triangulation and I_e the nodes of a finite element $e \in E$ then R is often defined as

$$R := \bigcup_{e \in E} I_e \times I_e$$

holds.

Sparse matrices are represented using special sparse matrix formats that stored only non-zero entries. On the other hand, sparse matrix equations are often solved with with iterative methods that can utilize the nature of the sparse matrix formats.

3.1.3.3 The Discrete System of Equations

The coefficients (a_{ij}) of the discrete system can be computed in an *element-oriented* way using the sets I_e and data g that are related with the elements $e \in E$.

$$a_{ij} \leftarrow \sum_{\{e \in E \,|\, i,j \in I_e\}} g(e, i, j). \tag{3.12}$$

For the Poisson problem 3.1 for example, the term $g(e, i, j)$ reads

$$\int_e \nabla\psi_i \cdot \nabla\psi_j \, dx, \tag{3.13}$$

where $\psi_i : \Omega \longrightarrow \mathbb{R}$ is defined as an approximating function for which

$$\psi_i(k) = \delta_{ik}$$

holds for each $k \in I$.

Iterative methods are often used for the solution of the approximating system of equations. Often preconditioned Krylov subspace methods[52] are utilized because they provide sufficient robustness for a wide class of discretized problems. Besides the preconditioning step the main components of iterative methods are

- matrix vector multiplications with the sparse matrix of Equation 3.11,
- vector additions and multiplications of vectors by scalars,
- inner products.

These numerical components have a high degree of data parallelism with respect to the node set I.

The convergence rate of Krylov subspace (and other iterative methods) depends on spectrum of the matrix. Preconditioning is a technique that tries to transform a linear system into an equivalent system with better spectral properties. A problem for parallel computing is that many traditional preconditioners such as the Gauss-Seidel relaxation method or incomplete factorization are highly sequential[52].

3.1.4 Mesh Generation and Mesh Restructuring

The decomposition of an N-dimensional continuum into a set of simple geometric objects is referred to as *mesh generation*. Mesh generation is an algorithmically demanding task, in particular, because certain quality requirements must be met[19, 83, 94]. In the case of triangulations for finite elements, for example, the angles of the triangles must not become too acute since this can have a negative impact on the stability of numerical algorithms.

While mesh generation can be used for a wide variety of applications, e.g. computer graphics, the principal applications of interest are finite element methods and other discretization methods.

Often not only one but several related meshes are used when solving finite element problems. Moreover, it is often preferable to start with a relative coarse mesh and later refine it to achieve a better resolution of certain features. Hereby knowledge gained by so far performed computation can be incorporated into the new mesh. Therefore one speaks of *adaptive mesh refinement*.

In order to simplify the process of generating adaptively refined meshes, often *local* refinement strategies are applied. Instead of generating a new mesh from scratch, some of the elements of the old mesh are subdivided into several elements.

Some applications require also *coarsening* of previously refined meshes, for example when shock waves must be represented. Here again *locally defined procedures* are often of advantage.

Mesh refinement and coarsening together are referred to as *mesh restructuring.*

3.1.4.1 Example: Red-Green Subdivision of Triangles

In the case of triangle elements, one possible subdivision strategy would be to decompose a triangle by bisecting its edges and create four smaller triangles as shown in Figure 3.6.

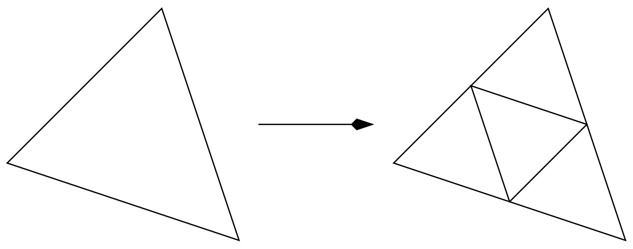

Figure 3.6: Regular subdivision of a triangle

This subdivision pattern is applied to a triangle if two or all of its edges are marked for refinement. Triangles created by this subdivision rule are referred to as *red.*

If only one edge has been marked then the triangle is subdivided into two triangles by connecting the midpoint of this edge with the opposite vertex. This is shown in Figure 3.7.

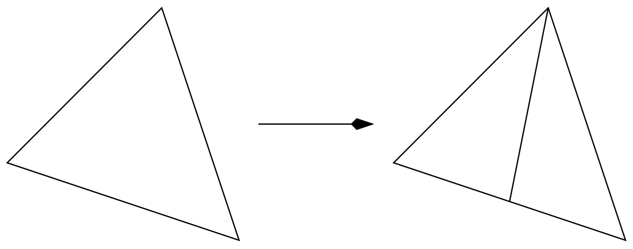

Figure 3.7: Irregular subdivision of a triangle

Triangles that are obtained by this rule are referred to as *green triangles.* The edge that is shared by two sibling green triangles is also referred to as *green.*

The point of the red-green subdivision rules is that Green triangles may not be further subdivided. This restriction ensures that the angles of a triangulation do not become too acute.

If an edge of a green triangle is marked for refinement then the this triangle and its (green) sibling are replaced by their parent red triangle to which the regular subdivision rule of Figure 3.6 is applied.

3.1.4.2 Example: Subdivision Rules for Tetrahedra

For tetrahedral meshes an analogous subdivision strategy would decompose a tetrahedron into four tetrahedra (see Figure 3.8) that are located at its vertices and a central octahedron that can be further decomposed into four other tetrahedra. This decomposition is, however, not uniquely determined since the subdivision of the inner octahedron depends on the selection of one of its diagonals[15]. Moreover, the resulting inner tetrahedra are in contrast to the for outer ones not similar to the original tetrahedron. This makes maintaining a high mesh quality more complicated.

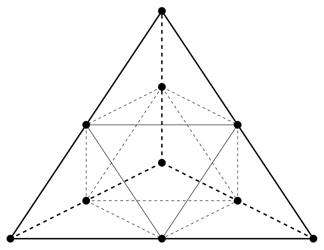

Figure 3.8: Regular subdivision of a tetrahedron

There are two subdivision patterns that deal with tetrahedra that have only a few edges marked for refinement (see Figure 3.9). The first one decomposes a tetrahedron into two tetrahedra whereas the other one decomposes it into four tetrahedra. These and other subdivision strategies for finite element meshes are more thoroughly discussed in[16, 19]

3.1.4.3 Closure of Restructuring Marks

Before the subdivision patterns presented in Sections 3.1.4.1 and 3.1.4.2 can be applied, there must be a *consistent* marking of the elements.

The problem is that an initial marking of edges of a triangulation may make it necessary to refine other edges as well. This process is called *clos-*

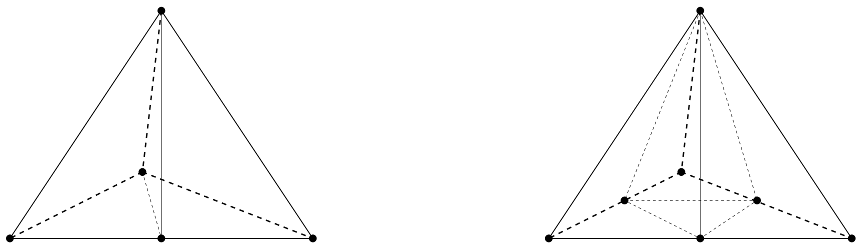
Figure 3.9: Irregular subdivision of a tetrahedron

ing the refinement and involves the repeated evaluation and update of the refinement marks. Figure 3.10 illustrates this. This holds analogously also for the subdivision rules of tetrahedra in Section 3.1.4.2.

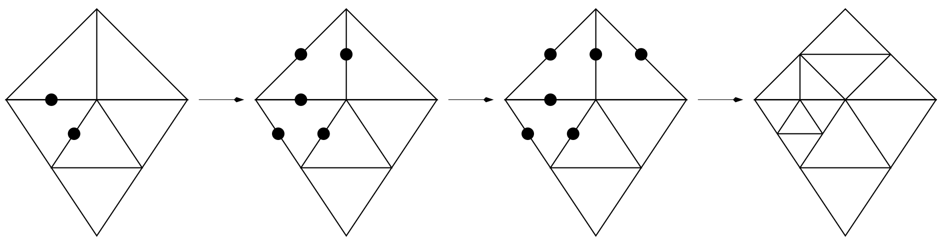
Figure 3.10: Closure of refinement marks and resulting subdivision

For the previously discussed triangle and tetrahedron subdivision rules, the refinement marks can be represented by a function m on the edge set E, that is.

$$m : E \longrightarrow \{0,\, 1\} \tag{3.14}$$

Here a value of 1 for $m(e)$ indicates that the edge $e \in E$ is to be refined. A value of 0 is interpreted as keeping the edge.

Closing subdivision marks is an *iterative* procedure. Starting with an initial marking $m^0 : E \longrightarrow \mathbb{N}$, it creates a *monotone increasing sequence* of refinement marks

$$m^0 \leq\ m^1 \leq \ldots \leq m^k. \tag{3.15}$$

Note that $m^i \leq m^{i+1}$ is argument-wise defined, that is, $m^i(e) \leq m^{i+1}(e)$ for all $e \in E$.

Note also that for the determination of m^{i+1} the edge marks of m^i must be evaluated for every element (triangle or tetrahedron). Thus, the *element-edge* relation plays a crucial role in this process. The evaluation can be performed independent on all elements of the mesh.

This monotone increase together with the fact that only a finite number of refinement markings exist for a given triangulation ensures that to sequence in Equation 3.15 reaches a maximum after finite steps. This maximum element represents a consistent refinement marking that can be subdivided.

3.1.5 Parallel Graph Algorithms

A graph G consists of a set V (its *vertices*) and relation $E \subset V \times V$ (its *edges*). Graphs are useful abstractions for the solution of many computer science problems.

Parallel graph algorithms have been investigated in[88]. The discussion there distinguishes between algorithms for *unweighted* graphs and algorithms for *weighted* graphs. Examples of unweighted graph algorithms are searching a graph and finding connected components. Examples that involve weighted graphs are shortest path algorithms and minimum spanning trees.

3.1.5.1 Example: The Bellman-Ford Algorithm

A graph problem that contains a high degree of data parallelism is the Bellman-Ford algorithm that solve the *single-source shortest-paths problem*[25]. For a given $G = (V, E)$ and a *weight* function $w : E \to \mathbb{R}$, this problem consists in finding the shortest path from a fixed source vertex $s \in V$ to every vertex of V.

The weight of a path $p = (v_0, v_1, \ldots, v_l)$, where (v_i, v_{i+1}) is an edge of G, is defined as

$$w(p) = \sum_{i=0}^{l-1} w(v_i, v_{i+1}).$$

The *shortest-path weight* between two vertices u, and v of G is defined as

$$\delta(u, v) = \begin{cases} \min_p\{w(p)\} & \text{over all paths from } u \text{ to } v. \\ \infty & \text{if no such path exists.} \end{cases}$$

A *shortest path* between two vertices u and v is defined as any path p for which $w(p) = \delta(u, v)$.

The Bellman-Ford algorithm solves the single-source shortest-path problem for general real-valued weights. Dijkstra's algorithm[25], on the other hand, can only be applied if the weights are non-negative. The Bellman-Ford algorithm indicates whether there are cycles with negative weights that can be reached from the source vertex. In this case no solution exists. If no cycle exist, the Bellman-Ford algorithm delivers the shortest paths and their weights.

The Bellman-Ford algorithm associates two attributes with each vertex $v \in V$, namely,

- the *shortest path estimate* $d : V \to \mathbb{R}$ that is an upper bound for the weight of the shortest path from s to v,
- the *predecessor* $\pi : V \to V$ in a path from s to v.

Initially, $d(v)$ is set to ∞ for all vertices except for s where it is set to 0. The initial value of $\pi(v)$ is `nil` for all vertices.

The essential component of Bellman-Ford and Dijkstra's algorithm is *relaxation over an edge* (u, v) and which reads

$$(d(v) > d(u) + w(u, v)) \Longrightarrow \begin{cases} d(v) & \leftarrow d(u) + w(u, v), \\ \pi(v) & \leftarrow u. \end{cases} \tag{3.16}$$

The data parallelism of Bellman-Ford is due to the fact that the relaxation step of Equation 3.16 can be performed simultaneously for all edges of G. The degree of parallelism is comparable with that of a matrix-vector multiplication. At most $|V| \cdot |E|$ relaxation steps must be performed to produce the shortest paths and their weights or to indicate that there are negative cycles that are reachable from s.

3.2 Properties of Data Parallel Applications

In § 3.1, several examples of data-parallel application have been investigated. The main source of data-parallel applications are computational problems that group around the numerical solution of partial differential equations (PDE). However, the field of graph algorithms or cellular automata also provide many data parallel applications.

The following list describes common properties of the algorithms and problem data that occur in these applications.

1. Algorithms are formulated in terms of (finite) sets and relations and functions that are defined on these objects.

2. Relations describe data dependences of the algorithms and are typically *sparse*.

3. Sets and relations do not change as frequently as the associated data. This observation, which is referred to as *relative stability* of the spatial structures, is essential when designing efficient abstractions for scientific applications and heavily influences the design of the Janus conceptual framework (see Chapter 4).

4. Sets and relations can be described by simple algebraic expressions or have to be represented by enumerating their elements.

One key observation of this domain analysis is the clear separation of objects that occur in data parallel applications into sets, relations, and functions that associated data with the both. The other important insight is the *relative stability* of the sets and relations. This has a great impact on the choice of data structures for these objects.

The relative stability however does not mean that sets and relations are considered completely fixed. The mesh-refinement algorithms (§3.1.4) show that in some circumstances *modifying algorithms* can be described by associating temporary predicates to mesh elements.

The following Chapter 4, presents the Janus conceptual framework for data structures and algorithms that rests on the insights gained during this domain analysis.

Chapter 4

The Janus Software Architecture

In this chapter the Janus software architecture for data parallel applications is presented. The design of Janus rests on the knowledge gained in during the domain analysis of data parallel application that has been performed in Chapter 3. Key requirements of the design are to define abstractions that allow the efficient representation of sets and relation and the attributes that are associated with them.

The description of the architecture uses the ideas of generic programming (§2.6) to formulate general syntactic, semantic, and complexity requirements that allow the construction of efficient components for this application domain. This conceptual framework provides the basic abstractions for the C++ template library *Janus* whose components are presented in chapter 5.

The conceptual framework of Janus rests on three major concepts, namely,

- Domain to represent finite sets, see Section 4.2,
- Relation to describe dependences of domain elements, see Section 4.4,
- Property Function to represent data that are associated with sets or relations, see Section 4.3.

The description of these abstractions will also reveal how the relative stability of sets and relations in data parallel applications is exploited.

Before describing the requirements of these abstractions, a short recapitulation of the most important terms of generic programming is given. Hereby the terminology of Musser's *concept web for generic programming* [77] is employed.

4.1 The Terminology of Concepts, Models, and Refinement

Generic programming has been informally defined by David Musser[78, 77] as *programming with concepts.* In the context of generic programming, the term *concept* names *families of abstractions* that are related by a common set of requirements.

Abstractions are data structures or algorithms. The requirements imposed on the abstractions regard their interfaces and their properties. More precisely: Interface requirements regard the *syntax* of objects and operations of an abstraction. Property requirements, on the other hand, state the *semantics* and resource constraints, for example, the complexity of the operation of an abstraction.

For example, the concept Container of the C++ standard library[98] requires that a container type `A` must provide a method "size"with the signature

```
A::size_type  A::size()
```

where the nested type `size_type` of `A` is an unsigned integer type. So far, this is mostly an interface requirement. A semantic requirement imposed on the `size` method is that it must return the number of elements in the container. Moreover, it is required that the complexity of `size` is linear in the number of elements of the container.

Models of a concept are abstractions that fulfill all requirements. The template classes `list` and `vector`, for example, satisfy all requirements of the concept Container and are thus models of this concept.

Usually, a concept description contains more than just a list of interface and property requirements. It can, for example, contain a list of concept it *refines.* A concept C' is said to be a refinement of the concept C if all requirements of C also belong to C'.

Refinement is not the only way to utilize other concepts in the description of a new concept. A concept description can also *use* other concepts to formulate its set of requirements. For example, the following description of Janus concepts uses various concepts of STL [98] to formulate certain requirements.

Notation: In the following chapters, concepts are set in sans serif font and start with a capital letter. Their required data types and members are set in `typewriter` font. Classes that are models of a concept are also set in `typewriter` font.

4.2 The Domain Concept

The concept Domain describes a *set* of n different objects. Figure 4.1 shows simple examples of finite sets. Note that arbitrary finite sets are considered not just two-dimensional point sets.

Figure 4.1: Examples of finite sets

Domain elements are numbered from 0 to $n-1$. Figure 4.2 shows a possible numbering of domain elements. A domain D is therefore a *finite sequence*

$$(d_0, d_1, \ldots, d_{n-1}) \tag{4.1}$$

with the additional requirement that the mapping $i \mapsto d_i$ is one-to-one.

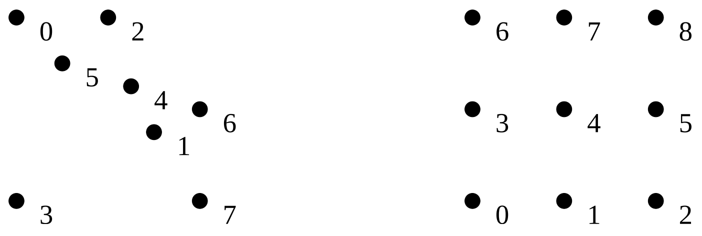

Figure 4.2: Examples of domains

The inverse function to $i \mapsto d_i$ is denoted by π, so it holds $i = \pi(d_i)$ for each i in the right-open[1] interval $[0, n)$.

$$\pi : D \longrightarrow [0, n) \tag{4.2}$$

is a bijection. The index i of a domain element d_i is also referred to as the *position* of this element with respect to the domain D.

4.2.1 Basic Domain Requirements

A type that is a model of Domain must provide the nested type `value_type` shown in Table 4.1.

Type	**Description**
`value_type`	The type of the objects of the domain. Similar to the STL[98] concept Container, the value type must fulfill the requirements of Assignable and Default Constructible[98].

Table 4.1: Nested types of the concept Domain

Table 4.2 shows the methods a domain must provide so that its elements can be queried. Note that all methods are *non-mutating*, i.e., calling any of these methods will neither change the elements of a domain nor their positions within the domain.

The complexity of element access (`operator[]`) is constant and is the same as that of a random access containers of STL[98] The `position` method on the other, is typically used to *search* for domain elements. The constraints for the `position` method have been chosen to allow using *binary search* for finding individual elements. This logarithmic complexity corresponds to that of the `find` method of Sorted Associative Container[98] like `std::set`.

Thus, Domain combines fundamental efficiency requirements of different container concepts of STL. The relationship of Janus domains and STL containers will further be investigated in §4.2.4. Requirements regarding the initialization of domains are presented and discussed in §4.2.2.

Table 4.3 gives the signature of constructors and related elementary methods a domain must provide. The methods of Table 4.2 are, as mentioned

[1]As in the description of the STL concepts and components [98], the mathematical notation $[a, b)$ denotes the range between a and b, where a is included but b is excluded from the range.

Method Name	**Description** (all methods are non-mutating)
Size	`size_t size()` Returns n, the number of elements. Complexity: $O(1)$
Element Access	`value_type operator[](size_t i)` Returns the ith element ($0 \leq i < n$). Complexity: $O(1)$
Position	`size_t position(const value_type& v)` Inverse method to `operator[]`. Returns `size()` if `v` is not an element. Complexity: Not worse than $O(\log n)$

Table 4.2: Methods required by the concept Domain

above, non-mutating. Using the assignment operator or the `swap` method from Table 4.3 it is, however, possible, to change the content of a domain. Table 4.3 shows the signature of constructors, the destructor, and assignment operations that are required for a model `X` of Domain.

Method Name	**Signature**
Default Constructor	`X::X()`
Copy Constructor	`X::X(const X&)`
Assignment Operator	`X& X::operator=(const X&)`
Destructor	`X::~X`
Swap	`X::swap(X&)`

Table 4.3: Additional methods of a model `X` of Domain

4.2.2 One-Phase Domain and Two-Phase Domain

The requirements formulated in the Tables 4.1 and 4.2 offer access to domain elements. Initialization of domains is not discussed there.

The concepts One-Phase Domain and Two-Phase Domain, provide a simple and general framework to handle the differences in the initialization of regular and irregular domains. Both concepts are refinements of Domain.

The requirements of the concepts One-Phase Domain and Two-Phase Domain rest on the relative stability of sets and relations as opposed to the associated data. The main benefit of these additional requirements is that *efficient* data structures and initialization procedures can be used for the representation of sets and relations.

4.2.2.1 Domains are Static Search Structures

The terminology of *one/two phases* has been motivated by *Static Search Structures*, a concept that has been defined (see [91]) as

> an Abstract Data Type with certain fundamental operations, e.g., initialize, insert, and retrieve. Conceptually, all insertions occur before any retrievals.

Such structures occur frequently in software system applications, where set members are inserted into the structure only once, usually during program initialization, and are generally not modified at run-time (see also *static sets* in [25]).

Simple (regular and static) spatial structures can be completely described at initialization time. The rectangular grid from Figure 4.2 is an example for One-Phase Domain. There is no need to insert individual elements into it. Grid elements can be retrieved immediately after the initialization is finished. Since they have only a retrieval phase but no insertion phase, they are referred to as *one phase structures* (see Figure 4.3).

Complex (irregular or dynamic) spatial structures, such as a finite element triangulation, usually cannot be completely described at initialization time. The left part of Figure 4.2 gives a simple example of an irregular domain.

For such domains an *insertion phase* provides a simple way to put individual elements or groups into them. Thus, an insertion phase would act as an *extended* initialization—Figure 4.3.

A very important constraint here is that domain elements can only be accessed when the insertion phase is completed and the so-called *access phase* has been entered. Therefore, they are referred to as *two phase structures*.

To be explicit: insertion and retrieval phases are clearly separated, i.e., no access is allowed in the insertion phase and vice versa. Remember, that no mutating operations can be performed during the access phase of a domain (see Table 4.2). Two-Phase Domain require a `freeze` method that marks the phase transition (see Table 4.5).

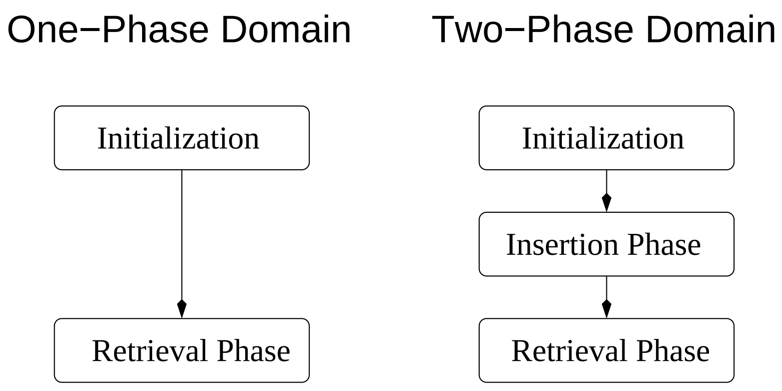

Figure 4.3: The phases of static search structures

4.2.2.2 Requirements

The concept One-Phase Domain does not add (nested) types or methods to the concept Domain. It only specifies that there is no insertion phase.

The concept Two-Phase Domain, on the other hand requires the nested type `insert_type` to represent objects that are to be inserted into a domain—see Table 4.4.

Type	**Description**
`insert_type`	The type of the objects to be inserted into the domain. Note that `insert_type` can be different from `value_type`. It can be used to insert (disjoint) *chunks* of `value_type` objects into the domain.

Table 4.4: Types required by Two-Phase Domain

The type `insert_type` is the argument of the the `insert` method in Table 4.5. After all elements have been inserted the domain is *closed* by calling the `freeze` method. This method marks the transition from the insertion phase to the retrieval phase there is the method `freeze`. To test whether a model of Two-Phase Domain has been frozen there is the method `frozen` (see Table 4.5).

Method	Description
Insert	`void insert(const insert_type& v)` Inserts the object(s) `v` into the domain. Each object may be inserted only once. This method may only be called before `freeze`. Complexity: see complexity of `freeze`.
Freeze	`void freeze()` Marks the transition from the insertion phase to the retrieval phase. The complexity of all calls to `insert` *and* the call to `freeze` is not worse than $O(n \log n)$. This allows to sort the elements of a domain which in turn enables the use of binary search in the method `position` (see Table 4.2).
Frozen	`bool frozen()` Returns true if and only if `freeze()` has been called. Complexity: $O(1)$

Table 4.5: Methods required by Two-Phase Domain

4.2.2.3 Thawing Frozen Domains

Naturally, the question arises, if it is possible to *freeze* a domain, is it also possible to *thaw* it?

Mesh adaptation, for example, is an example where a `thaw` method could be used to express *modification* of existing domains. For the Janus framework, however, the answer is *no* and the main reasons for this decision are *simplicity of design* and *efficiency.*

To justify this decision a general modification algorithm is considered. Table 4.6 shows the different approaches to express *modification* of a domain X. The left column shows the main steps for the case with `thaw` the right column if only `freeze` is available.

It is acceptable and often conceptually clearer to express *domain modification* by creating a new domain, copying the needed data to the new domain, and ignoring the unnecessary data. Table 4.6 also indicates that in addition to a `thaw` method a `remove method` would be necessary. Erasing data from a data structure, however, can be expensive as the `erase` method of the `vector` container of the STL shows.

More important, relations (§4.4) and associated data depend on the posi-

Modification with `thaw`	**Modification without `thaw`**
Thaw X	Declare temporary domain Y
Remove unnecessary elements	Insert needed elements from X
Insert new elements	Insert new elements
Freeze X again	Freeze Y
	Swap X and Y

Table 4.6: Domain modification with and without `thaw`

tion of domain elements. These positions, however, are likely to change when modifying a domain. Thus, relations and associations of a domain have to be changed as well. Therefore, we claim that a non-thaw approach is easier to manage.

PETSc, the Portable, Extensible Toolkit for Scientific Computation[6], also uses the so-called "rebuild data structure" approach for numerical problems that require mesh refinement.

The decision to provide no `thaw` method is also supported by data base technology. The Standard Query Language (SQL) provides an `ALTER` statement to change minor aspects (e.g. adding columns) of a table. However, for substantial changes it is recommended to create a new table in the desired format and to copy the content of the old table into the new one (see [33]).

4.2.3 The Distributed Domain Concept

By definition, Domain describes a finite sequence of objects $d_0, d_1, \ldots, d_{n-1}$ that can be accessed through `operator[]` and searched by `position` (see Table 4.2).

The concept Distributed Domain is used to indicate when a domain is spread over a group of processes. The background here are computer architectures with *distributed memory.* In order to take into account the much higher cost of remote memory accesses, a fundamental requirement of Distributed Domain is that all domain methods introduced so far (see Table 4.2 and 4.5) act *local* to the (sub) domain object for which they are invoked.

4.2.3.1 The Janus Process Model

Janus rests on the same *static* process model as MPI [97]. This means that processes are the fundamental computational unit. Each process consists of a thread of control that is executed in a separate address space. A Janus application is executed by a collection of processes that form a *communication context*. In MPI , communication contexts are referred to as *communicators*. As in MPI, the elements of a communication context of p processes are indexed from 0 to $p-1$. The number of parallel processes and their process identification numbers can be queried by the global functions `processes` and `process` that are shown in Table 4.7.

Global Function	**Description**
Number of Processes	`size_t processes()` Returns the number of parallel processes.
Process Identification	`size_t process()` Returns the unique process number of the calling process.

Table 4.7: Global functions for process identification.

The processes are executed in an SPMD style. There is no mechanism for loading code onto processors, or assigning processes to processors. Also there is no explicit support for multi-threading in Janus. However, operations on domains or relations can utilize several threads.

4.2.3.2 Subdomains of Distributed Domains

Let there be a communication context of p processes. A *distributed domain* D is a collection of p *mutually disjoint* domains D_i

$$D = \bigsqcup_{i=0}^{p-1} D_i. \tag{4.3}$$

The domain D_i is referred to as the i-th *subdomain* of D (see Figure 4.4).

Let n_i be the size of the subdomain D_i then the sum

$$n = \sum_{i=0}^{p-1} n_i \tag{4.4}$$

is referred to as the *total size* of the domain D.

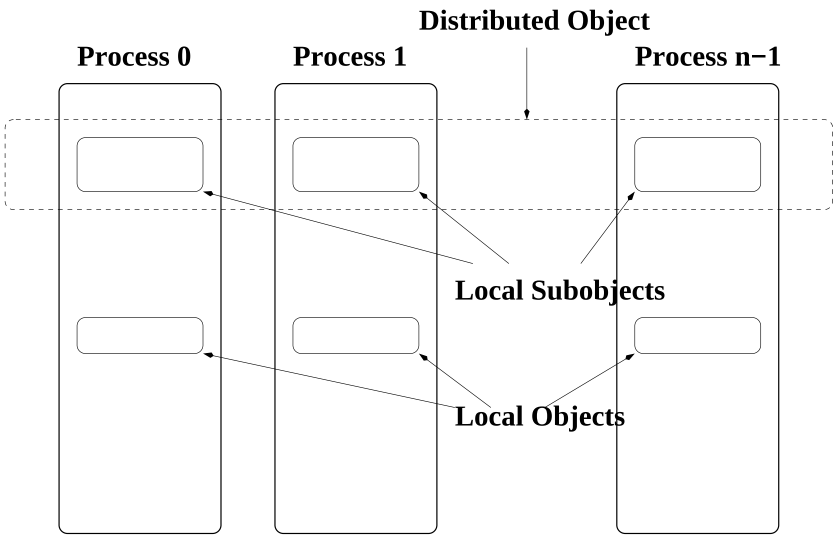

Figure 4.4: Local sub-objects of distributed objects.

4.2.3.3 Global and Local Positions

The position of an elements with respect to a subdomain D_i is referred to as *local position*. The vary in the range $[0, n_i - 1)$. A local position is unique only with respect to its subdomain.

In order to maintain the important property of *one-to-one correspondence* of domain elements and their positions, a *global position* j_g is associated with the j-th element in process i. The simple relation between local and global positions involves only the sizes n_i of the subdomains and reads

For this the concept of a *global position offset* of the subdomain D_i is defined as the partial sum of subdomain sizes up to process number $i - 1$, that is

$$b_i = \sum_{k=0}^{i-1} n_k. \tag{4.5}$$

Note that in the series $(b_0, b_1, \ldots, b_p)$ (p the number of processes) b_0 equals 0 whereas b_p equals the total size n of Equation 4.4.

The definition of a global position j_g of the j-th element of subdomain D_i then simply reads

$$j_g = b_i + j. \tag{4.6}$$

Figure 4.5 shows local and global positions of the domains from Figure 4.2 for a communication context consisting of two processes. Local positions are given as lower indices, global positions as upper ones.

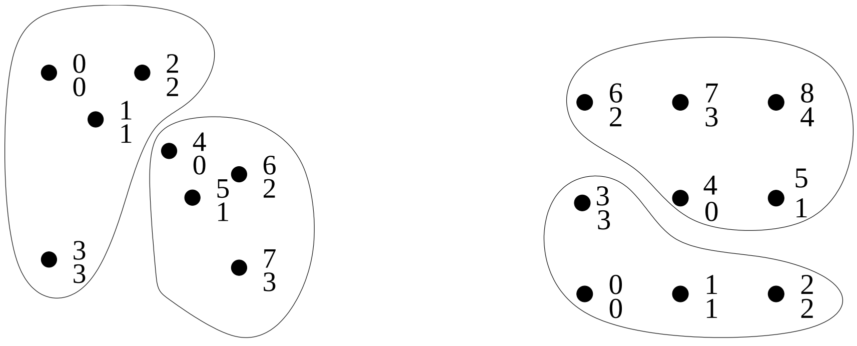

Figure 4.5: Examples of local and global positions

For each subdomain D_i, the sequences $(n_0, \ldots, n_{p-1})$ and $(b_0, b_1, \ldots, b_p)$ are stored in a *descriptor subobject* that has the type `descriptor_type`. Table 4.8 shows that the descriptor object of a distributed domain can be queried by the method `descriptor()`. The type `descriptor` itself and related utilities are explained in Section 5.6.2.

Method Name	**Description**
Descriptor	`const descriptor_type& descriptor() const` The type `descriptor_type` provides methods to access size information of subdomains—see Table 5.13. Complexity: $O(1)$

Table 4.8: The `descriptor` method of Distributed Domain

4.2.3.4 The Distributed Two-Phase Domain Concept

For Models of Distributed Domain that are also models of Two-Phase Domain there is a method `insert_at` that is described in Table 4.9.

The method `insert_at` is the main primitive of Janus to specify mapping information when inserting elements into a distributed two-phase domain. Note that due to the semantics of Two-Phase Domain the transfer of `v` into the domain of process `p` can be delayed until returning from `freeze`. This emphasizes the usefulness of the Two-Phase Domain concept for the efficient implementation of irregular/dynamic distributed scientific applications.

Method Name	**Description**

Global Insert	`void`
	`insert_at(const insert_type& v, size_t p)`
	Inserts `v` into the subdomain of process `p`.
	See Table 4.4 for a description of `insert_type`.
	Complexity: See complexity of `freeze` in Table 4.5.

Table 4.9: The `global_insert` method of Distributed Two Phase Domain

4.2.4 Relationship of Domain and Container

There are some differences between the Janus Domain concept and the STL concept Container.

A container, in the sense of the STL [98], is an object that *stores* objects of a certain type ("its elements"). A container provides iterators to traverse and access its elements. A Janus domain can be a container but it does not have to be since it is not required that it stores its elements. The elements of regular domains, for example a rectangular grid, can be described much more efficiently by simple constraints.

Domains whose elements have to be explicitly stored must be models the STL Container concept. Having the access requirements of a domain in mind, it is not surprising that if a model of Domain is also a model of Container then it is a model of Random Access Container. However, only *non-mutating* access is allowed (see Table 4.10).

4.3 The Property Function Concept

The concept Property Function describes a collection of objects of a type `T` that are associated with the elements of a domain `D` or a relation `R` (see §4.4). The reader shall think for example of pressure and velocity values that are related to grid components and to matrix coefficients associated with relations.

Mathematically, these objects are *functions* on sets or relations. If D is a domain then a function

$$f : D \longrightarrow T \tag{4.7}$$

Type/Method	**Description**
`size_type`	An unsigned integer type that can hold the number of elements of the container.
`difference_type`	A signed integer type.
`const_iterator`	A read-only Random Access Iterator.
`const_reference`	A type that behaves as a const reference to the domain's value type.
`const_iterator begin()`	Returns an iterator to the first element of the domain.
`const_iterator end()`	Returns an iterator pointing one past the last element in the domain.
`const_iterator find( const_reference v)`	Finds `v` or returns `end()` if `v` i s not an element of the domain. This method shall have the same complexity as `position`.

Table 4.10: Nested types and methods of container domains.

where associates objects of type T with the elements of D.

If n is the size of the domain D then using the function π of Equation 4.2 the function

$$f \circ \pi^{-1} : [0, n) \longrightarrow T \tag{4.8}$$

is considered—see Figure 4.6.

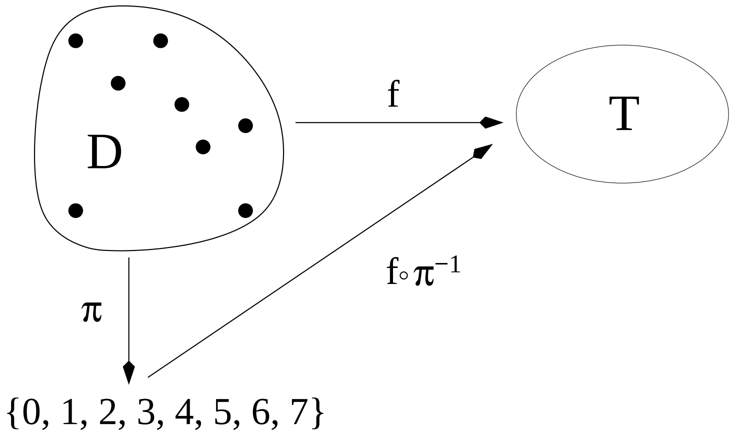

Figure 4.6: Property functions and domain positions

The name *property function* goes back to the concept Property Graph of BGL (Boost Graph Library[67]). Property functions that are associated with a domain or a relation are also referred to as *domain functions* or *relation function*, respectively.

The concept Property Function requires only `operator[](size_t)` to provide random access to n objects of `T`—see Table 4.11.

Method Name	**Description**
Element Access	`T operator[] (size_t i)` Returns the ith element ($0 \leq i < n$) where n is the size of the underlying domain. Complexity: $O(1)$

Table 4.11: Random access required by the concept Property Function

This approach allows to use various simple and efficient data structures to represent property functions. For example, it is possible to use one-dimensional C and Fortran arrays or the standard container template class `std::vector<T>` to represent a property function with value type `T`. An advantage of using such minimal requirements to represent data on domains is that it is very easy to interoperate with other libraries and languages.

Property Functions on Distributed Domains

For a distributed domain $D = \bigsqcup_{i=0}^{p-1} D_i$ a property function $f : D \to T$ is considered as a family of property functions

$$f_i : D_i \to T, \qquad 0 \leq i < p.$$

This means that a property function f is distributed according its underlying domain D.

This design decision is different from the approaches taken by the C++ template libraries POOMA[61] and Blitz++[105]. Both libraries provide as basic abstractions *multi-dimensional array* classes. In Janus, on the other hand, dimensions and other *spatial* information are expressed by the underlying domains or relations. For this reason, models of Property Function are not restricted to rectangular arrays and can also describe data on irregular domains.

4.4 The Relation Concept

The concept Relation is based on the mathematical concept of a relation between two sets X and Y. In mathematics, a relation R is a subset of the cross product $X \times Y$, i.e., a set of ordered pairs (x, y).

In Janus, a relation R between two domains X and Y describes a set of ordered pairs $(x, y) \in X \times Y$. However, since domain elements have unique (global) positions, elements of a relation between two domains are represented as *pairs of integers* and not as pairs of domain elements. Therefore the `position` methods π_X and π_Y of the domains X and Y, respectively, will play a crucial role.

If m and n are the number of elements of the domains X, respectively Y then the functions

$$\begin{aligned} \pi_X &: X \longrightarrow [0, m) \\ \pi_Y &: Y \longrightarrow [0, n) \end{aligned}$$

assign to each domain element its position. The requirements of Domain ensure that the position function is a bijection.

Therefore for each relation $R \subset X \times Y$ the following associated relation $R' \subset [0, m) \times [0, n)$ can be defined as

$$R' = \pi_X \circ R \circ \pi_Y^{-1}.$$

In other words, a relation between two domains is represented as the set of the pairs of positions of the elements in the relation

$$R' = \{(\pi_X(x), \pi_Y(y)) \,|\, (x, y) \in R\}\,.$$

The Janus concept Relation has been devised for *sparse* relations. A relation $R \subset X \times Y$ is called *sparse* if for the cardinality of the involved sets holds

$$|R| = O(|X| + |Y|). \tag{4.9}$$

Sometimes a stronger requirement is imposed, namely, that there is a positive number C so that

$$|R_i| < C \qquad \text{for all } i \in [0, m) \text{ where} \qquad R_i := \{j \,|\, (i, j) \in R'\}\,. \tag{4.10}$$

Analogously to the concept `domain`, a relation R' is considered as a *sequence of position* pairs

$$(i_0, j_0), (i_1, j_1), \dots, (i_k, j_k). \tag{4.11}$$

This, however, does not define the *internal* representation of a relation object, rather special *sparse storage formats* are deployed. The point of these formats is to provide a *compact* description of the relation and to allow the *efficient* implementation of basic linear algebra operations such as sparse matrix-vector multiplication or solution of triangular systems—see the Draft Sparse BLAS standard [80].

4.4.1 Basic Relation Requirements

There are similarities but also notable differences between the concepts Relation and Domain.

Analogously to Domain, the concept Relation requires that, there is a nested `value_type` which however is defined as "pair of objects of `size_t`" (see Table 4.12).

Type	Description
`value_type`	This is a typedef for `jns::pair_size_type` (Table 4.13).

Table 4.12: Nested types of the concept Relation

Type	Description
`jns::size_vector`	Typedef for `std::vector<size_t>`
`jns::pair_size_type`	Typedef for `std::pair<size_t,size_t>`
`jns::pair_size_vector`	Typedef for `std::vector<pair_size_type>`

Table 4.13: Some globally defined auxiliary types

As for Domain (see Table 4.2), there is a `size` method to query the number of pairs of a relation. Table 4.14 presents this method together with methods that return information about the size of the involved domains.

Analogously to Table 4.2, Relation requires a `position` method that returns the index of an element. There is, however, no random access (i.e. through `operator[]`) to its elements. Rather there are a couple of methods that allow accessing relation elements (x, y) through the first entry x. This

Method	**Description** (all methods are *const* methods)
Size	`size_t size()` Number of elements of the relation. Complexity: $O(1)$
Size 1	`size_t size1()` Size of the first factor of a relation. Complexity: $O(1)$
Size 2	`size_t size2()` Size of the second factor of a relation. Complexity: $O(1)$

Table 4.14: Basic query methods for size information required by Relation.

is similar to the widely used *compressed row storage* (CRS) sparse matrix scheme[80], where the index x is referred to as "row entry" whereas y is referred to as "column entry".

However, providing access methods in the style of CRS format does not mean that a model of Relation actually uses the CRS format to store its elements. It only means that the elements of a relation can be efficiently accessed through the methods of Table 4.15. In order to avoid a too close binding to the CRS format the terms *first entry* and *second entry* instead or *row* and *column* are used. Note however that the relation member function `size1()` of Table 4.14 plays a distinguished role in the description of the methods in Table 4.15.

4.4.2 One-Phase Relation and Two-Phase Relation

Similar to the concept Domain, there are the concepts One-Phase Relation and Two-Phase Relation. These concepts formulate requirements for the initialization of relation objects. Since a relations deals with the position of its domain elements a relation can only be constructed *after* its domains have been completely initialized.

As in the case of Domain, there are clearly separated insertion and retrieval phases (§4.2.3).

One-Phase Relation are completely initialized by calling a constructor. Two-Phase Relation, on the other hand, have `insert` methods that allow the insertion of individual elements *before* the method `freeze` has been called. These methods are described in Table 4.16.

The `freeze` method transforms the set of inserted pairs into sparse matrix format. This enables both a *compact representation* and an *efficient implementation* of data transfer operations (§4.4.4). In the case of *distributed*

Method	**Description** (all methods are *const* methods)
Position	`size_t position(const value_type& v)` Index of `v`. Returns `size()` if `v` is not an element.
Size of Row	`size_t size(size_t i)` The number of elements that correspond to first index i. It must hold $i \in [0, \texttt{size1}())$. Complexity: $O(1)$
Element Access	`value_type value(size_t i, size_t k)` Returns the k-th element of the i-th row. It must hold $i \in [0, \texttt{size1}())$ and $k \in [0, \texttt{size(i)})$. Complexity: $O(1)$
Position	`size_t position(size_t i, size_t k)` Equivalent to `position(value(i,k))`. Complexity: $O(1)$
First Entry	`size_t first(size_t i)` This is equivalent to `value(i,k).first` for all $k \in [0, \texttt{size(i)})$. Complexity: $O(1)$
Second Entry	`size_t second(size_t i, size_t k)` Equivalent to `value(i,k).second`. Complexity: $O(1)$

Table 4.15: Access methods required by Relation.

Method Name	**Description**
Freeze	`void freeze()` Completes the initialization of the relation. Complexity: Not worse than $O(n \log n)$, n is the number of elements of the relation. As in Table 4.5, this includes the work for element insertion.
Frozen	`bool frozen()` Returns true if and only if `freeze()` has been called. Complexity: $O(1)$

Table 4.16: Methods required by Two-Phase Relation

relations (§4.4.3), this includes the negotiating buffer sizes for remote operations *before* they are actually used. This again shows the suitability of two-phase structures for parallel scientific applications.

No methods for insertion of relation elements are specified since these are highly dependent on the concrete relation type. However, contrary to the case of Two-Phase Domain, relation elements may be inserted into a relation more than once— see Table 4.5

4.4.3 Distributed Relation

As in the case of Domain, there is a refinements of Relation to efficiently handle the constraints of distributed memory architectures. The concept Distributed Relation operates with pairs of *global* positions. Row-oriented representation of relations (see §4.15) also means that a pair (i, j) is hold at the same process where the position i is kept. Thus, the mapping of a relation $R \subset D^1 \times D^2$ is determined by the mapping of X, or in other words

$$R' = \bigsqcup_{k=0}^{p-1} \left\{ (i,j) \,|\, i \in D^1_k \right\} . \tag{4.12}$$

Since by definition the mapping of relation is determined by its first relation factor, there is no need to explicitly specify mapping information and consequently there is no `insert_at` method (see Table 4.9) for distributed two-phase relations.

Similar to Table 4.8, there is a method `descriptor()` that returns a reference to a descriptor sub-object of a distributed relation—see Table 4.17.

Method	Description
Descriptor	`const descriptor_type& descriptor() const` Returns distribution information for the relation. See also Table 5.13. Complexity: $O(1)$
Descriptor 1	`const descriptor_type& descriptor1() const` Returns distribution information for the first domain. See also Table 4.8. Complexity: $O(1)$
Descriptor 2	`const descriptor_type& descriptor2() const` Returns distribution information for the second domain. See also Table 4.8. Complexity: $O(1)$

Table 4.17: Methods required by Distributed Relation.

4.4.4 A Generic Interface for Data-Parallel Operations

The concept Relation must provide expressive methods to perform typical matrix-vector and related operations of data-parallel applications.

Consider a relation $R \subset X \times Y$ and a property function b on the domain Y. As in Equation 4.10, for each position i of X the set R_i denotes the positions of Y that are related with i with respect to R. For the property function b the set $b(R_i)$ is defined as the image of b restricted to R_i, that is

$$b(R_i) := \{b_j \,|\, j \in R_i\}\,. \tag{4.13}$$

The general idea is to provide methods that allow for each position i of X to *access* the elements of $b(R_i)$.

Example Here is an example for the five-point stencil of Equation 3.5 on Page 41. If b is a function on a two-dimensional grid G and i is the position of a grid point $(x, y) \in G$ then $b(S^5_i)$ is the set

$$\Big\{b(x, y-1),\, b(x, y+1),\, b(x, y),\, b(x-1, y),\, b(x+1, y)\Big\}.$$

Here, $S^5 \subset G \times G$ denotes the five-point stencil.

The basic idea is to require that a relation provides a nested `accessor` type—similar to the nested iterator types of STL container. The accessor type must be template class that is parameterized over the type of data to be accessed. If `R` is a model of Relation then the declaration of the associated accessor type reads:

```
template<T>
class R::accessor;
```

The accessor type is used to declare accessor objects that are bound to a particular relation and the data to be accessed

```
void foo(const R& r, const std::vector<double>& b)
{
    R::accessor<double> acc(r, b.begin(), b.end());
    // ...
}
```

Here `b` must be a property function that is associated with the second domain of the relation object `r`. In particular, the following assertion

```
assert(r.size2() == b.end() - b.begin());
```

must be fulfilled. In order to access individual elements of `b` the methods of Table 4.18 are used.

Method	Description
Pull	`void pull()` Prepares the direct access to the data of the property function `b` to which the accessor has been bound. In the case of a distributed relation, this includes the transfer of the non-local data of $\bigcup_i b(r_i)$. Complexity: Not worse than $O(\|R\|)$
Get	`T get(size_t i, size_t k)` Accesses the element of the property function `b` with the position `r.second(i,k)`. Complexity: O(1).

Table 4.18: The methods `pull` and `get` of the accessor type of a relation

A problem of these general operations is that they cannot completely reflect the underlying sparse matrix format of a relation. As a consequence, they might be less efficient for standard sparse matrix operations such as matrix-vector multiplication. Therefore Relation also provides member methods that provide efficient implementations of these operations. These methods are presented in the following subsection.

4.4.5 Data-Parallel Member Operations

In order to provide efficient support for important sparse matrix operation, the concept Relation provides the `pull_reduce` and `pull_matrix` methods. The point of these methods that they utilize the internal sparse matrix format of the relation. Moreover these methods are highly customizable by the user. Both the binary reduction operation(s) and the action performed on the reduced value can be specified by the library user.

4.4.5.1 The `pull_reduce` Method

The first reduction scheme uses the sum of the values in $b(R_i)$

$$\sum_{j \in R_i} b_j \tag{4.14}$$

or the result of a user-defined binary reduction operation $\oplus$

$$\bigoplus_{j \in R_i} b_j \tag{4.15}$$

for argument reduction.

The interfaces of the version of `pull_reduce` that use `operator +` as binary reduction operator is shown in Figure 4.7.

```
template<typename T, typename Visitor>
void R::pull_reduce(const T* t, Visitor v);
```

Figure 4.7: Signature of `pull_reduce`

The template parameter `T` is the type of the values that are stored in the property function represented by the pointer `t`.

The template parameter `Visitor` denotes a *visitor type*, that is, an object `v` of this type is used in the sense of the *visitor design pattern*[37] for each pair

$$\left(i, \bigoplus_{j \in R_i} b_j\right), \tag{4.16}$$

where i runs over all positions of the second domain of R. The requirements for a visitor type are specified in Table 4.19. Janus provides several standard visitor types that are explained in §5.3.3.

Name	Description
Apply	`void Visitor::operator()(size_t i, value_type r)` The action performed for all pairs in Equation 4.16.

Table 4.19: Requirements for a visitor type

The semantics of `pull_reduce` for a model `R` of Relation is described using nested accessor type introduced in the previous subsection. Note that the code in Figure 4.8 does not necessarily reflect the actual implementation.

The variable `reduce` is default-initialized that is by the result of the expression `T()`. The C++ standard specifies that for built-in arithmetic

```
template<typename T, typename Visitor>
void R::pull_reduce(const T* t, Visitor v)
{
  // bind accessor to relations and data
  typename R::accessor<T> access(*this, t, t+r.size2());

  access.pull(); // transfer data
  for(size_t i = 0; i < size1(); ++i) {
    T reduce = T();
    for(size_t k = 0; k < size(i); ++k)
      reduce += access.get(i, k); // access data
    v(i,reduce);
  }
}
```

Figure 4.8: Semantics of `pull_reduce` defined through `accessor`

types, such as `int` or `float`, the value of default initialization is `0` of the respective type[99].

The complexity of a `pull_reduce` operation is $|X|$ applications of the argument reduction and the same number of applications of the visitor object `v`. Note that the visitor object is passed per value.

The second version of `pull_reduce` (see Figure 4.9)allows to specify a user-defined binary reduction operation [2] $\oplus$ and an initial value for it. The template parameter `BinaryOp` declares the user-defined operation $\oplus$ of Equation 4.15. The fourth function argument `init` of `pull_reduce` is the initial value for the reduction operator.

4.4.5.2 The `pull_matrix` Method

The second argument reduction scheme is motivated by matrix-vector multiplication, that is, it uses the value

[2]The binary operation must be (at least theoretically) associative and commutative because for the sake of an efficient implementation the order in which the reductions are performed is *not* specified. Standard floating point types do not fulfill this requirement. However, the differences that occur from different orders in which the reduction operations are performed are ignored in this definition.

```
template<typename T, typename Visitor, typename BinaryOp>
void R::pull_reduce(const T* t, Visitor v, BinaryOp op,
                                  T init);
```

Figure 4.9: Signature of extended version of `pull_reduce`

$$\sum_{j \in R_i} m_{ij} \cdot b_j \tag{4.17}$$

or the result of a user-defined binary reduction operations $\oplus$ and $\otimes$

$$\bigoplus_{j \in R_i} m_{ij} \otimes b_j \tag{4.18}$$

for argument reduction.

As in the case `pull-reduce` there are two overloaded versions of `pull_matrix`. The interface of the simple version is shown in Figure 4.10.

The template parameter `T` is the type of the matrix coefficients and the vector elements. The requirements of Table 4.19 hold also for the visitor type `Visitor` in Figure 4.10.

```
template<typename T, typename Visitor>
void R::pull_matrix(const T* m, const T* t, Visitor v);
```

Figure 4.10: Signature of `pull_matrix`

The semantics of `pull_matrix` for a model `R` of Relation is described using the nested `accessor` type from Section 4.4.4. As for the case of `pull_reduce` the code in Figure 4.11 does not necessarily reflect the actual implementation. A major difference to the code in Figure 4.8 is the use of the second `position` method from Figure 4.15 to access the corresponding matrix coefficient.

The semantics of `pull_reduce` uses the binary functions `operator +`[3] and `operator *`. As in Figure 4.8, the default value of type `T` is used.

[3]Strictly speaking, it is using `operator +=` which implies that its definitions must correspond to those of `operator +` and the assignment operator.

```
template<typename T, typename Visitor>
void R::pull_matrix(const T* m, const T* t, Visitor v)
{
    // bind accessor to relations and data
    typename R::accessor<T> access(*this, t, t+r.size2());

    access.pull(); // transfer data
    for(size_t i = 0; i < size1(); ++i) {
        T reduce = T();
        for(size_t k = 0; k < size(i); ++k)
            reduce += m[position(i,k)] * access.get(i, k);
        v(i,reduce);
    }
}
```

Figure 4.11: Semantics of `pull_matrix` defined through `accessor`

In the more elaborate version of `pull_matrix` (see Figure 4.12), different types `M` and `T` for the matrix coefficients and vector elements are allowed. Moreover, user-defined binary reduction operators $\oplus$ and $\otimes$ can be specified. As in the case of the more elaborate version of `pull_reduce` (Figure 4.8), an initial value for the reduction can be specified.

```
template<typename M, typename T, typename Visits,
         typename Mult, typename Plus>
void R::pull_matrix(const T* m, const T* t, Visitor v,
                    Mult mult, Plus plus, T init );
```

Figure 4.12: Signature of `pull_matrix`

4.5 Conclusions

The Janus software architecture rests on three major concepts. The concepts Domain and Relation formulate various requirements to describe (distributed) sets and their relations. The concept Property Function, on the other hand, formulates only very few and simple requirements do describe data that are associated with sets or relations. It is this clear separation of *sets and relation*, on the one hand, and associated data on the other hand, that distinguishes the Janus from related architectures such as PETSc[6].

Models of Property Function can therefore be very simple data types. basically one-dimensional arrays, pointers, and random access iterators fulfill these requirements. As it will be seen in the following chapters, this simplicity leads to high performance. It also enables interoperability with other scientific software packages.

A remarkable feature of both the Domain and Relation concepts is that they have the semantics *static search structures*. This design decision is justified by the *relative stability* that was recognized as a result of the domain analysis performed in Chapter 3. This insight allows to devise flexible and yet efficient models of these concepts—see the following chapter. In particular, as Chapter 7 will show, it makes highly dynamic data structures, such as tree or linked lists dispensable for the implementation of adaptive finite element and related numerical methods.

The requirements of Relation follow state-of-the-art interfaces for sparse matrix operations. In particular, matrix-vector multiplications are supported. However, efficient access through relations to individual elements is also supported. In Chapter 7 and Chapter 8, this will turn out as a key advantage when engineering parallel adaptive finite element methods and parallel graph algorithms.

Chapter 5

Components of Janus

In Chapter 4, the major concepts of the Janus architecture have been presented. This architecture consists of a set of requirements for data types and algorithms for data parallel applications. In this chapter the models, that is, template classes, template algorithms, and additional utilities of Janus are briefly presented. The emphasis is on presenting concrete data structures that can be applied in various fields of data parallel scientific applications. However, this chapter does not describe how specific applications are developed using the Janus components. This is discussed in Chapter 7. Likewise, implementation and portability issues of the components are not addressed in this chapter. These problems are discussed in the following chapter 6.

The (template) classes presented in this chapter are models of the concepts Domain §5.2 and Relation §5.3 discussed in §4. Thus, the description of these classes can be restricted to the features they offer *in addition* to those required by the concepts.

The small set of algorithms presented in §5.5 consists on the one hand of reduction operations for scalar types (§{refaccumulator), operations on property functions on domains (§5.5.2.2) or relations (§5.5.2.3). A unifying feature of these algorithms is that they *accumulate* individual values and that they provide an explicit terminate operation (`freeze`). This makes them very useful in the context of a *distributed* computing environment.

The other set of algorithms are parallel versions (§5.5.3) of appropriate STL algorithms, for example `std::count`. Note that this approach differs significantly from HPC++[60] which aimed to provide distributed version of all STL containers, parallel iterators, and parallel versions of STL algorithms.

Janus provides no components that are models of Property Function. The requirements of this concept (§4.3) are so minimal that they can be easily satisfied by existing standard classes. Section §5.4 lists some alternatives.

Section §5.6 presents auxiliary types and functions that simplify the task of writing application programs which can run efficiently in a shared and distributed environment.

5.1 Janus as a Program Family

The conceptual framework of chapter §4 and the components presented in this chapter form the *Janus program family* in the sense of Parnas' definition in §2.2. Throughout the rest of this thesis the term *Janus* is used in the sense of *Janus program family*.

The concepts of Janus represent the common properties of the components. The components of this chapter provide solutions for particular application requirements. Here, the main distinguishing factors are different requirements regarding their dynamic behavior. This can be seen in Figure 5.1 which shows the Domain and Relation classes of Janus.

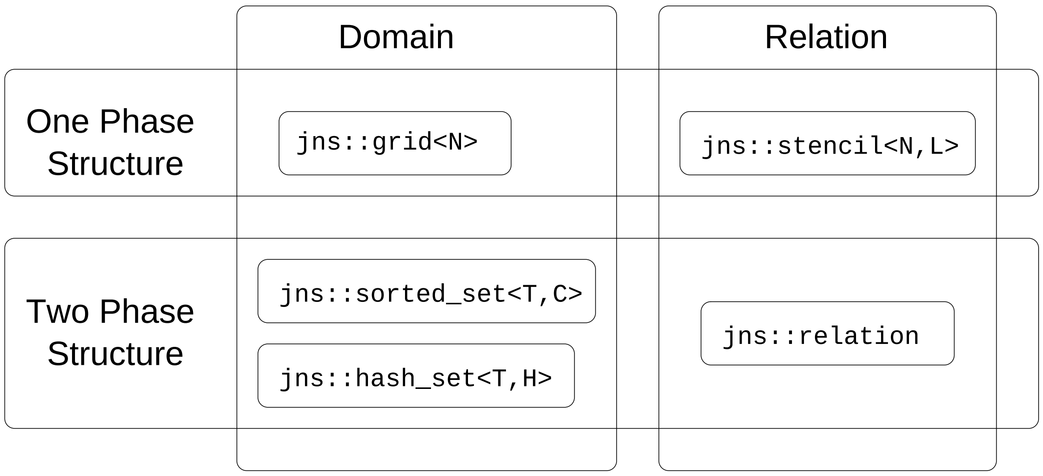

Figure 5.1: Overview of Janus domains and relations

Figure 5.1 depicts the relationship of the Janus components to the Janus concepts. In particular, it can be seen whether a model of Domain is a model of One-Phase Domain or Two-Phase Domain. Note that all Janus components belong to the namespace `jns`. A more detailed description of these domain and relation classes is given in §5.2 and §5.3.

5.1.1 Local and Distributed Components

Figure 5.1 does not specify whether the components shown there are models of the concepts Distributed Domain or Distributed Relation.

In fact, all components in Figure 5.1 can be used *pure locally* or as *distributed objects*. Using these components simplifies the task of writing programs that work both for non-distributed memory and distributed memory platforms. It also emphasizes that there are customized family members for specific application scenarios.

At the same time it is important that clients of Janus components do not have to pay for unwanted services. In particular, components that are to be deployed shall not be burdened by distributed services.

Janus uses a simple trick to provide uniform *names* to components in local and distributed environments. As Figure 5.2 suggests, there are two sub-namespaces of `jns`, namely the namespaces `local` and `distributed`.

```
namespace jns
{
    namespace local
    {
        class A { /* .. */ };
    }
#ifdef JANUS_LOCAL
    using local::A;
#endif

#ifdef JANUS_DISTRIBUTED
    namespace distributed
    {
        class A { /* .. */ };
    }
    using distributed::A;
#endif
}
```

Figure 5.2: Janus namespaces for local and distributed components

Local components are defined in the sub-namespace `local`. If the preprocessor flag `JANUS_LOCAL` is defined then the class `jns::local::A` is exported into the surrounding namespace `jns`, that is, it can be accessed as `jns::A`.

Distributed components, on the other hand, are only defined if the preprocessor flag `JANUS_DISTRIBUTED` is set. In this case the class `jns::distributed::A` can be accessed also as `jns::A`. At the same time, the local component `jns::local::A` can be used as well.

5.2 Models of Domain

In this section the models of Domain that has been shown in Figure 5.1 are explained in more details. In particular, all involved Janus concepts, the template parameters of the classes, and important additional features of the classes are presented.

5.2.1 Models of Two-Phase Domain

The templates `sorted_set` §5.2.1.1 and `hash_set` §5.2.1.2 are models of Two-Phase Domain that implement different strategies to store and access domain elements. Common to both template classes is that they are also models of Container(see §4.2.4).

5.2.1.1 The `sorted_set` Template

The template parameters and the non-trivial constructors of `sorted_set` are shown in Table 5.1:

```
template<typename T, typename Compare = std::less<T> >
class  sorted_set {
public:
    typedef T  value_type;

    sorted_set(Compare comp = Compare());
    // ...
};
```

Table 5.1: Parameters and constructor of `sorted_set`

The type parameter `T` is the value type of this model of Domain. The second parameter `Compare` is a comparison function that must implement a *strict weak ordering* in the sense of STL[98]. The comparison object can be specified through a constructor. By default `std::less` is used as comparison function.

Upon freezing, the inserted elements are sorted according to the comparison function and stored in an internal random access container so that they can be searched later efficiently with binary search[1]. Note that the complexity requirements of the `position` method of Domain matches those of the STL binary search algorithms.

Using `sorted_set` to represent a two-phase domain has the advantage of being optimal with respect to memory usage. No additional storage is necessary to represent all elements of a domain. The drawback is the relatively slow logarithmic time of binary search.

Applications that desire for different time/space tradeoffs the template `hash_set` can be of advantage.

As discussed in 5.1, the class `sorted_set` can be (depending on configuration) be a model of Distributed Domain. In this case it provides the method `insert_at` (Table 4.9) to specify the mapping of the element to be inserted.

5.2.1.2 The `hash_set` Template

The template parameters and non-trivial constructor of `hash_set` is shown in Table 5.2.

The type parameter `T` is the value type of this model of Domain. The second parameter `Hash` is a hash function, that is Unary Function[98], which accepts objects of type `T` and returns a `size_t`. The hash function can be specified in the constructor of `hash_set`.

Upon freezing, the inserted elements are put into an internal hash table. This makes the complexity of method `hashed_domain::position` dependent on quality of the hash function and the distribution of inserted elements. In case both factors work together well then method `position` has an average constant complexity.

As for the template `sorted_set`, the class `hash_set` can be a model of Distributed Domainand thus provide the method `insert_at` of Table 4.9.

[1]To be more precise, the template function `std::lower_bound` from the family of binary search algorithms is used.

```
template<typename T, typename Hash>
class  hash_set {
public:
    typedef T  value_type;

    hash_set(Hash hash = Hash());
    // ...
};
```

Table 5.2: Parameters and constructor of `hash_set`

5.2.2 Models of One-Phase Domain

Janus provides only one model of One-Phase Domain, namely the template class `grid` §5.2.2.1. Rectangular grids are a class of simple and commonly used domains. They are, however, by no means the only possible model of One-Phase Domain. One could easily provide classes that represent general polytopes of $\mathbb{Z}^n$ or regular substructures of rectangular grids.

5.2.2.1 The `grid` Template

The class `grid<N>` is a model One-Phase Domain that describes an N-dimensional rectangular grid. The template parameters and non-trivial constructor are shown in Table 5.3.

```
template<size_t N>
class grid {
public:
  typedef  boost::array<int,N>  value_type;

  grid(value_type a, value_type b);
  // ...
};
```

Table 5.3: Parameters and constructor of `grid`

The declaration of the value type of grid is used the fixed-size array template of the Boost[28] library. The constructor with the two arguments a and b of value_type creates the grid

$$\prod_{i=0}^{N-1} [a_i, b_i) \subset \mathbb{Z}^N. \tag{5.1}$$

Depending on the configuration of Janus (§6.1), grid is also a model of Distributed Domain and provides a pre-defined mapping.

5.3 Models of Relation

In this section the Janus classes that are models of Relation are presented. As in the case of model of Domain, the presentation distinguishes between models of Two-Phase Relation §5.3.1 and models of One-Phase Relation §5.3.2. Currently, Janus provides for each of these refinements of relation only one model that correspond to often used sparse matrix formats. Depending on the configuration of Janus these models also satisfy the requirements of relation.

5.3.1 Models of Two-Phase Relation

Janus' standard model of Two-Phase Relation is the relation template.

5.3.1.1 The relation Template

The class relation of Table 5.4 implements the *compressed row sparse* format[80]. This class is *not* a template class. However, it provides template member function pull_reduce and pull_matrix that are required by Relation.

The class relation has a template constructor which allows to initialize the size query size1 and size2 methods of Table 4.14 with the respective sizes of the domain objects x and y. This also implies that a relation can only be initialized if the corresponding domains are completely initialized.

Only domain positions can be inserted into relation. For this the insert(size_t, size_t) method can be used. In order to insert domain elements into a relation the algorithm class conversion_collector of §5.5.1.3 can be used.

```
class relation {
public:
  typedef jns::pair_size_type  value_type;

  template<typename X, typename Y>
  relation(const X& x, const Y& y);

  void insert(size_t, size_t);

  void freeze();

  // ...
};
```

Table 5.4: Parameters and constructor of `relation`

The initialization of a relation must be finished explicitely by calling the `freeze` method.

5.3.2 Models of One-Phase Relation

Janus' only model of One-Phase Relation is the `stencil` template which is presented in Section 5.3.2.1.

5.3.2.1 The `stencil` Template

The template class `stencil` is to be used in conjunction with the `grid` domain class of §5.2.2.1.

The two template parameters of `stencil` represent the dimension of the underlying grid and the size of the stencil.

In order to initialize a stencil, not only the grid and the *offsets* of the stencil must be specified but also a *subgrid* on which the offsets can be safely applied. This is important because for boundary points of the grid the stencil might not be correctly defined. The corresponding constructor is shown in Table 5.5.

```
template<size_t N, size_t L>
class  stencil  {
  typedef  jns::pair_size_type  value_type;
  typedef  grid<N>::value_type  grid_value_type;
  typedef  boost::array<grid_value_type,L> offset_type;

  stencil(const grid<N>& g, offset_type offsets,
          grid_value_type a, grid_value_type b);
  // ...
};
```

Table 5.5: Parameters and constructor of `stencil`

5.3.3 Visitor Classes

The template member methods `pull_reduce` and `pull_matrix` (§4.4.5) expect *visitor* objects that perform operation with the values reduced by the relation. Table 4.19 states that the operation is performed by `operator()(size_t i, T t)`.

Table 5.6 provides a small set of visitor template classes that solve some often occurring standard cases.

Assign	`template<typename T> class assign_visitor` Assigns the reduced value at position `i` of a property function represented by the type `T*`.

Binary Function	`template<typename T, typename Op = std::plus<T> >`
	`class binary_function_visitor`
	Combines the reduced value with the value at position `i` of a property function represented by the type `T*` using the binary function `Op`.

Table 5.6: Standard visitor classes of Janus

The class `assign_visitor` can be used to implement a generic matrix-vector multiplication $y \leftarrow A \cdot x$ where the matrix $A = (R, m)$ is a pair of a relation R and matrix coefficients $m : R \longrightarrow T$ (see Figure 5.3).

The `binary_function_visitor` could be used for a matrix-vector multiplication with a subsequent binary operation, for example, in an expression

```
template<typename Rel, typename T>
void multiply(const Rel& R, const T* m, const T* x, T* y)
{
    R.pull_matrix(m, x, jns::assign_visitor<T>(y));
}
```

Figure 5.3: Generic matrix-vector multiplication

life $y \leftarrow y - A \cdot x$ (see Figure 5.4). This example also suggest that more complex (generic) version of matrix-vector multiplication could be easily implemented by providing appropriated visitor classes.

```
template<typename Rel, typename T>
void multiply1(const Rel& R, const T* m, const T* x, T* y)
{
    jns::binary_function_visitor<T, std::minus<T> > v(y);
    R.pull_matrix(m, x, v);
}
```

Figure 5.4: Using `binary_function_visitor` to customized matrix-vector multiplication

5.4 Models of Property Function

As mentioned in §4.3, Janus does not provide models of the concept Property Function. The reason is that the simple requirements of this concepts can be easily fulfilled by pointers, and standard classes such as `std::vector` and `std:valarray`. User-defined vector classes can also easily cooperate with Janus as long as they provide a mean to access their data as specified in Table 4.11.

5.5 Algorithms

The algorithms provided by Janus can be classified into

- *Collectors* §5.5.1 which simplify the initialization of domains/relations and associated property functions.
- *Accumulators* §5.5.2 support computations whose result depends on the evaluation of accumulated values,
- *Parallel STL* algorithms §5.5.3 which are wrappers for STL algorithms with parallel semantics.

5.5.1 Collectors

Collectors are data structures that *temporarily* store pairs of keys and data objects. They are used for the initialization of domains/relations and associated property functions. The problem hereby is that before a property function can be manipulated its underlying domain/relation must have been initialized. Another important aspect handled by collectors is that they *transfer* the pairs to other processes where they can be handled locally. Thus, collectors play a crucial role in handling with explicitly or implicitly mapping information.

Collectors are used similar to two-phase domains (see Section 4.2.2):

1. Elements are *inserted*,
2. The collector is *frozen*, and
3. *Finally*, the inserted elements can be *accessed*.

Common requirements of constructors are the types and methods given in Table 5.7.

5.5.1.1 The `collector` Template

The template class `collector` of Figure 5.5 stores pairs of objects of type `K` and `T`.

Pairs can be inserted using the following two `insert` methods. There are also the methods `insert_at` which indicate that the inserted element of

Type/Method	Description
`key_type`	The key type of the elements in the collector.
`data_type`	The type of data stored in the collector.
`value_type`	Defined as `std::pair<key_type,data_type>`.
`void freeze()`	Marks the end of insertions.
`size_t size()`	Number of locally accessible inserted elements.
`const value_type& operator[](size_t)`	Access of elements.

Table 5.7: Nested types and methods required of collectors.

```
template<typename K, typename T>
class collector {
public:
 typedef K key_type;
 typedef T data_type;
 typedef std::pair<key_type,data_type> value_type;

 void insert(const key_type& k, const data_type& t);
 void insert(const value_type& v);

 void insert_at(const key_type& k, const data_type& t, size_t p);
 void insert_at(const value_type& v, size_t p);
};
```

Figure 5.5: The `collector` template class

`value_type` is transferred to process `p`. This transfer operation, however, is known to be completed only after the `freeze` of Table 5.7 has been called.

This class is particular useful for initializing (distributed) domains and associated property functions.

5.5.1.2 The `relation_collector` Template

The template class `relation_collector` of Figure 5.6 stores pairs of `pair_size_type` (see Table 4.13) and `T`. This class provides a template constructor that binds the collector to the domains of a relation.

If the pair `(k,t)` is inserted into the relation collector then `k.first` and

```
template<typename T>
class relation_collector {
public:
  typedef pair_size_type  key_type;
  typedef T               data_type;
  typedef std::pair<key_type,data_type> value_type;

  template<typename X, typename Y>
  relation_collector(const X& x, const Y& y);

  void insert(const key_type& k, const data_type& t)
  void insert(const value_type& v)
};
```

Figure 5.6: The `relation_collector` template class

`k.second` must be *global* positions of the domains `x` and `y`, respectively. In according with the mapping of relations (see Section 4.4.3) the inserted pair `value_type(k,t)` is transferred to the process of the global position `k.first` with respect to the domain `x`. As in §5.5.1.1, all transfer operations are known to be completed only after `freeze` has been called.

5.5.1.3 The `conversion_collector` Template

The template class `conversion_collector` of Figure 5.7 stores pairs of `size_t` and `T` objects. The template parameter `X` must be a model of Domain or Relation. The constructor argument `x` must be a *completely initialized* domain or relation.

Inserted pairs of `X::value_type` and type `T` are transformed into pairs of `size_t` and `T`. The `size_t` object is the *local* position of the `X::value_type` object with respect to the the process to which this object belongs. The transformed pair is transferred to this process. The transformation and the transfer are completed after `freeze` has been called.

A particular useful application of the class `conversion_collector` shows the implementation of the Janus algorithm `n_relation_insert` in Figure 5.8.

Let `x` and `y` be two domains of type `X` and `Y`, respectively. Let `G` be a

```
template<typename X, typename T>
class conversion_collector {
public:
  typedef size_t  key_type;
  typedef T       data_type;

  conversion_collector(const X& x);

  void insert(const typename X::value_type& e,
              const data_type& d)
};
```

Figure 5.7: The `conversion_collector` template class

unary function object[98] whose *argument type* is `X::value_type` and whose *result type* is `boost::array<Y::value_type,N>` for some integer constant `N`. For each element `a` of `x` all elements of `gen(a)` must be elements of the domain `y`.

The `n_relation_insert` algorithm converts the pairs

$$\left\{(a, G(a)_j) \,|\, a \in x, 0 \le j < N\right\} \tag{5.2}$$

into their respective positions and inserts them into the relation object `r`.

The implementation of the algorithm shown in Figure 5.8 use the `conversion_collector` with the template parameters `Y` and `size_t`. For each position `i` of `x` and each element `w` of `gen(x[i])` the pair `(y,gi)` is inserted in the collector object. Hereby, `gi` is the *global position* of `i`. After freezing the collector the contains pairs of elements `(j,gi)` where `j` is a local position of an element of `y`. This local position is transformed into its global counterpart `gj` and the pair `(gi,gj)` is inserted into the relation.

5.5.2 Accumulators

Accumulators are algorithms whose result depends on the evaluation of groups of values. In contrast to STL algorithms, the values need not to belong to containers or iterator ranges.

Another difference is that accumulator algorithms are implemented as *classes* not as functions. The idea is to provide an *insertion* phase to *submit*

```
template<typename X, typename Y, typename G, typename R>
void
n_relation_insert(const X& x, const Y& y, G gen, R& r)
{
  conversion_collector<Y,size_t> rep(y);
  for(size_t i = 0; i < x.size(); ++i)
  {
      const typename G::result_type w = gen(x[i]);
      const size_t gi = jns::global_position(x,i);
      for(size_t k = 0; k < w.size(); k++)
          rep.insert(w[k],gi);
  }
  rep.freeze();
  for(size_t i = 0; i < rep.size(); ++i)
  {
      const std::pair<size_t,size_t> v = rep[i];
      assert(v.first < y.size());
      r.insert(v.second,jns::global_position(y,v.first));
  }
}
```

Figure 5.8: The `n_relation_insert` algorithm

values to the accumulator and an explicit termination function. Using classes instead of functions to implement such algorithms is natural because it is much easier to handle the any state related to the different phases. This approach is of course inspired by the insertion and access phases of Two-Phase Domain and Two-Phase Relation of §4.2.2 and §4.4.2, respectively. As in the case of domains and relations, the *two-phase approach* to deal with groups of values is particular useful in distributed programming environments.

Janus provides accumulators for different situations.

- The `accumulator` of §5.5.2.1 is used for the *reduction* of individual values.
- The `position_accumulator` of §5.5.2.2 can be used to combine values with elements at the *position* of property functions.
- The `relation_accumulator` of §5.5.2.3 should be used when values are

to be combined with elements of property functions given by relation elements (that is, pairs of integers).

5.5.2.1 The `accumulator` Template

The purpose of the `accumulator` template is the *reduction* of accumulated scalar values. The public interface of this template class is shown in Table 5.8.

```
template<typename T, typename BinaryFunc = std::plus<T> >
class accumulator {
public:
  accumulator(T init = T(), BinaryFunc bf = BinaryFunc());

  void insert(T t);

  T freeze();
};
```

Table 5.8: Public interface of `accumulator`

The two template parameters specify the scalar the type of the objects to be accumulated and the binary operation to be used thereby. If no binary operation is specified than the values are *added* to an initial value. The initial value can be specified in the constructor of `accumulator`.

Values that are to be reduced are *inserted* into the accumulator. The result of the reduction of all values—regardless in which process they were inserted—is determined and returned by calling the `freeze` method. After calling `freeze` no values may be inserted anymore. Moreover `freeze` may be called only once.

This somewhat unusual syntax for scalar reduction operation is analogous to the two phases of Two-Phase Domain. The explicit separation of insertion and access phases provides, as in the case of domains and relations, several opportunities for an efficient implementation on parallel machines. In particular, communication or synchronization that is necessary to determine the reduction can be deferred until `freeze` is called.

5.5.2.2 The `position_accumulator` Template

The template class `position_accumulator` whose public interface is shown in Table 5.9 can be used to manipulate values of a property function that is associated with a domain *or* a relation. The property function is represented by a pair of *random access iterators*.

```
template<typename X, typename T,
                     typename BinaryFunc = std::plus<T> >
class position_accumulator {
public:
  typedef size_t  key_type;
  typedef T       data_type;

  template<typename Iterator>
  position_accumulator(const X& x,
                       Iterator first, Iterator last,
                       BinaryFunc op = BinaryFunc());

  void insert(key_type i, const data_type& v);

  void freeze();
};
```

Table 5.9: Public interface of `position_accumulator`

For manipulating the property function a user-defined binary operator of type `BinaryFunc` is used. By default, `std::plus<T>` is used as binary function. Note that `T` is also the value type that is associated with the iterator type template parameter `Iter` in Table 5.9.

The constructor of `position_accumulator` takes a reference to the domain/relation object of type `X` and a pair of iterators that describes the property function. In order to manipulate the property function, a pair of global position `i` and an object of `T` must be inserted into the accumulator.

It is guaranteed that upon return from the accumulator's `freeze` method, all inserted values have been combined with the values at the corresponding positions. Hereby the binary operation `BinaryFunc` has been used.

5.5.2.3 The `relation_accumulator` Template

The template class `relation_accumulator` can be used to manipulate values of a property function that are associated with a relation of type `R`. The public interface of this template class is shown in Table 5.10.

```
template<typename R, typename T,
         typename BinaryFunc = std::plus<T> >
class relation_accumulator {
public:
  typedef  pair_size_type  key_type;
  typedef  T               data_type;

  template<typename Iterator>
  relation_accumulator(const R& r,
                       Iterator first, Iterator last,
                       BinaryFunc op = BinaryFunc());

  void insert(size_t i, size_t j, const data_type& v);
  void insert(const key_type& k,  const data_type& v);

  void freeze();
};
```

Table 5.10: Public interface of `relation_accumulator`

The property function is represented by a pair of *random access iterators* of type `Iter`. For manipulating the property function a user-defined binary operator of type `BinaryFunc` is used. By default, `std::plus<T>` is used as binary function.

The constructor of `relation_accumulator` takes a reference to the relation object of type `R` and a pair of iterators that describes the property function. In order to manipulate the property function, triples of global position `i`, `j`, and an object of `T` must be inserted into the accumulator.

It is guaranteed that upon return from the accumulator's `freeze` method, all inserted values have been combined with the values at the corresponding positions. Hereby the binary operation `BinaryFunc` has been used.

5.5.3 Parallel STL Algorithms

Janus provides a parallel implementation for some of the STL algorithms[98]. The focus is on algorithms that return *scalar* values that are the result of a *reduction operation*. The signature of some of the suitable STL algorithms (`accumulate`, `inner_product`, or `count`) are shown in Table 5.11.

```
template <typename InputIterator, typename T>
T accumulate(InputIterator first, InputIterator last,
             T init);

template <typename InputIterator1,
          typename InputIterator2, typename T>
T inner_product(InputIterator1 first1,
                InputIterator1 last1,
                InputIterator2 first2, T init);

template <typename InputIterator,
          typename EqualityComparable>
std::iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last,
      const EqualityComparable& value);
```

Table 5.11: Examples of parallel STL algorithms

Parallel versions of these algorithms can be easily implemented by using Janus' `reduce` template function of Table 5.12. The function `reduce` returns its first argument when called in a non-distributed context (see also 6.1). When called in the context of p parallel processes, that is, for the distributed tuple $(t_0, t_1, \ldots, t_{p-1})$, then `reduce` returns the value

$$\bigoplus_{i=0}^{p-1} t_i. \tag{5.3}$$

Here, $\oplus$ denotes the template argument `BinaryFunction` which must be a binary operation. The second version of `reduce` uses $+$ as binary operation.

Note that as the other Janus components, the parallelized versions of the algorithms of the Tables 5.11 and 5.12 are defined in the namespace `jns`.

```
template<typename T, typename BinaryFunction>
T reduce(T t, BinaryFunction op = BinaryFunction());


template<typename T>
T reduce(T t);
```

Table 5.12: Signature of `reduce`

5.6 Utilities

This section describes various minor utilities of the Janus framework. These utilities encompass, on the one hand, functions to initialize and query the runtime environment (§5.6.1) and, on the other hand, the `descriptor` class and related functions (§5.6.2).

5.6.1 Runtime Environment

5.6.1.1 The `initialize` Function

The function `initialize` sets up the runtime environment of a Janus program. It must be called before any Janus object is created. This implies that there can be no Janus objects in global scope. Arguments that are passed to `initialize` depend on the underlying communication system (e.g. MPI).

The general form of this function is

```
jns::initialize(int& argc, char**& argv);
```

5.6.1.2 The `finalize` Function

The function `jns::finalize()` shuts down the Janus runtime environment. This function is called without arguments.

5.6.1.3 The Functions `process` and `processes`

The function `jns::processes()` returns the number of parallel processes that participate in a Janus applications.

```
size_t jns::processes();
```

In case of a sequential configuration of Janus `processes` returns 1.

The function `jns::process` returns the *identification number* of the current process.

```
size_t jns::process();
```

This number is between 0 and `processes()-1`.

5.6.2 The Class `descriptor_type` and Related Functions

The type `descriptor` has been introduced in Section 4.2.3 for the description of *distributed* domains and relations. The main methods (which are all non-mutating) of this auxiliary type are shown in Table 5.13.

Method	**Description**
`size_t size(size_t k)`	Returns n_k, the size of subdomain D_k.
`size_t offset(size_t k)`	Returns b_k, the offset of subdomain D_k.
`size_t global_position (size_t j)`	Returns global position for local position j.
`size_t global_position (size_t k, size_t j)`	Returns global position of local position j of subdomain D_k.
`size_t process(size_t g)`	Returns process of global position g.
`pair_size_type process_position(size_t g)`	Returns pair of process id and local position of global position g. For `pair_size_type` see Table 4.13.

Table 5.13: Public methods of `descriptor_type`

Closely related to the type `descriptor` are several *global* template functions. They are, strictly speaking, not really necessary but they provide an additional abstraction that smooth the difference between different configurations of Janus (see also Section 6.1).

5.6.2.1 Global Size of a Domain or Relation

The template function

```
template<typename X>
size_t jns::size(const X& x)
```

returns the *global* size of a domain/relation `x`. In case of a non-distributed object `x` this functions returns the value `x.size()`.

5.6.2.2 Conversion of Global and Local Positions

The function `jns::global_position` converts a local position `i` of a domain/relation `x` into its global position.

```
template<typename X>
size_t jns::global_position(const X& x, size_t i)
```

If `i` is not a local position this function returns the global size of `x`. For a non-distributed domain/relation this function acts as the identity.

The function `jns::local_position`, on the other hand, converts a global position `g` of a domain/relation `x` into its local counterpart.

```
template<typename X>
size_t jns::local_position(const X& x, size_t g)
```

If `g` is not the global position for any position `i` of `x` then this function returns `x.size()`. As in the case of `global_position`, this function acts for a non-distributed domain/relation as the identity.

5.6.2.3 The Load of a Distributed Object

In order to check the quality of the distribution of a domain the following metric is used. Let D be a distributed domain that consist of the subdomain $D_0, \dots, D_{p-1}$. If n_i is the number of elements of the subdomain D_i then the *load* L of the distributed domain D is defined as

$$L := p\frac{\max\{n_0, \dots, n_{p-1}\}}{\displaystyle\sum_{i=0}^{p-1} n_i}. \tag{5.4}$$

This means that an even distribution has a load of 1. An uneven distribution is characterized by a load greater than 1.

The function `jns::load` returns the load of a domain/relation of type `X`.

```
template<typename X>
double load(const X& x);
```

In case of a non-distributed object `x` this function return the value 1.0.

5.7 Conclusions

This chapter presented the most important models of the concepts of Chapter 4. It has been shown that the conceptual framework of Janus is flexible enough to support, on the one hand, rectangular index spaces and typical stencil relations on them, and arbitrary sets and general sparse relations on the other hand as well. At the same time, these domain and relation classes are available for distributed and non-distributed application scenarios. Thus, the Janus framework and its data structures support a broad range of data parallel applications.

Since most of the data structures of the Janus library are C++ template classes they can be easily and efficiently configured for user-defined data types. This avoid a major drawback of libraries such as PETSc[6] that for historic reasons are written in the C programming language. At the same time, Janus components can easily mixed with components that are written in C and Fortran.

The key for Janus' ability to support both regular and irregular applications rests on the idea on one-phase and two-phase data structures and its clear separation of insert and access operations. Many of the algorithms discussed in this chapter also follow this idea. As it will be seen in Chapter 7, this is particularly suited for complex initialization procedures of data parallel applications. Another remarkable feature of the Janus template library are the parallel versions of simple STL algorithms.

Chapter 6

Porting Janus to Specific Parallel Platforms

The chapters 4 and 5 described the general requirements and corresponding components of the Janus framework. This chapter discusses with various implementation aspects of the Janus components and their configuration for different parallel architectures. JADE, a parallel package for Janus, is introduced and details of its MPI implementation are presented. The performance of Janus on top of MPI will be evaluated in Chapter 8.

Section 6.1 gives an overview on the internal classification and configuration possibilities of the Janus components.

Section 6.2 discusses the reuse of data structures and algorithms from (standard) libraries such as STL[98] and Boost[28]. Additional components are developed where those libraries do not provide the necessary functionality.

Section 6.3 introduces JADE which stands for *Janus Distributed Engine.* JADE provides various layers of abstractions for distributed memory architectures and constitutes a *port package* for Janus. It is explained how the JADE constructs can be implemented using the Message Passing Interface[97] (MPI).

Section 6.4 discusses issues of a portable shared memory implementation of the Janus components. The focus there is on OPENMP[102] which is an accepted portable programming model for shared-memory platforms.

6.1 Configuration of Janus

The conceptual framework of Janus supports independent configurations of its components for *distributed-memory* and *shared-memory* architectures. As Figure 6.1 shows, basically all user-visible components are available for distributed and local-memory operation modes. In Section 5.1 it has been demonstrated how the sub-namespace `distributed` and `local` of the namespace `jns` can be activated using the C++ preprocessor.

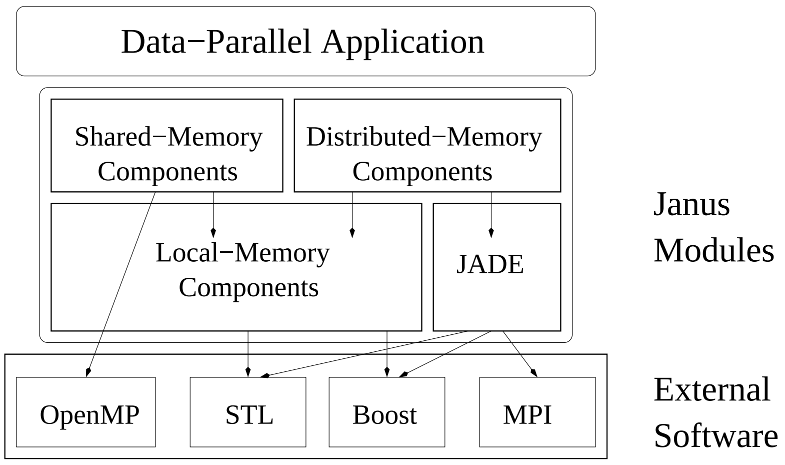

Figure 6.1: Dependence of Janus Modules

There are of course differences regarding the interfaces and semantics of the components in the different namespaces. For example, mapping information can be specified for the `sorted_set` or `hash_set` template class when configured for distributed memory.

Another difference concerns the `size()` method of the concept Domain. In case of a distributed configuration, it returns only the number of elements that are contained in the subdomain of the calling process (see §4.2.3). However, these differences can be overcome to some extend by using the auxiliary functions from Section 5.6.2.

6.2 Basic Implementation Aspects

The major part of the Janus Modules in Figure 6.1 are the *local core components*. They comprise most of the components in the namespace `jns::local`.

Due to the close relation between Janus and STL, there happens a lot of reuse here. For example the STL *sort* algorithm is used in the implementation of `sorted_set::freeze()`.

The implementation of most of the domain/relation classes of Janus uses the `vector` container to store elements. Thus the implementors and maintainers of Janus can rely on the dynamic memory management system of the C++ standard library. Standard function objects for comparison, logical and numerical operations are also frequently used.

The Boost[28] library collection is another base on which the implementation of Janus rests. Here the fixed-size array class `boost::array` and the class `boost::tuple` for *inhomogeneous* arrays are used, for example to define the value type of `grid<N>` (see Section 5.2.2.1).

One important algorithm that is missing both in STL and Boost is *counting sort*[25]. This algorithm allows sorting in *linear* time, provided the elements to be sorted can be considered as a *bounded* set of integers. Within Janus, counting sort is used to sort the keys of the two-phase hash table `hash_set` (§5.2.1.2) Counting sort is also used in the Jade, the distributed abstraction layer of Janus (§6.3), in order to initialize message buffers efficiently.

6.3 Jade—Janus Distributed Engine

Jade separates the implementation of distributed Janus components from lower level parallel communication software.

Jade itself consists—as shown in Figure 6.2—of three layers. The different layers reflect different functionalities. It starts with simple functions for starting, finishing, and querying the run time environment. Then it continuous with simple various data movement operations and ends with two-phase data movement classes. Figure {jade-layers also shows Janus components whose implementation use the Jade-components.

The first layer provides Jade counterparts for the utility functions of the Janus runtime environment presented in Section 5.6.1. The second layer consists of the three functions `jade::collect`, `jade::exchange_size`, and `jade::exchange_buffer` which are all *collective operations* that represent various send/receive patterns. The term *collective operation* is used in the sense of MPI [97] to describe data movement and computation operations that are performed by all processes in a computation.

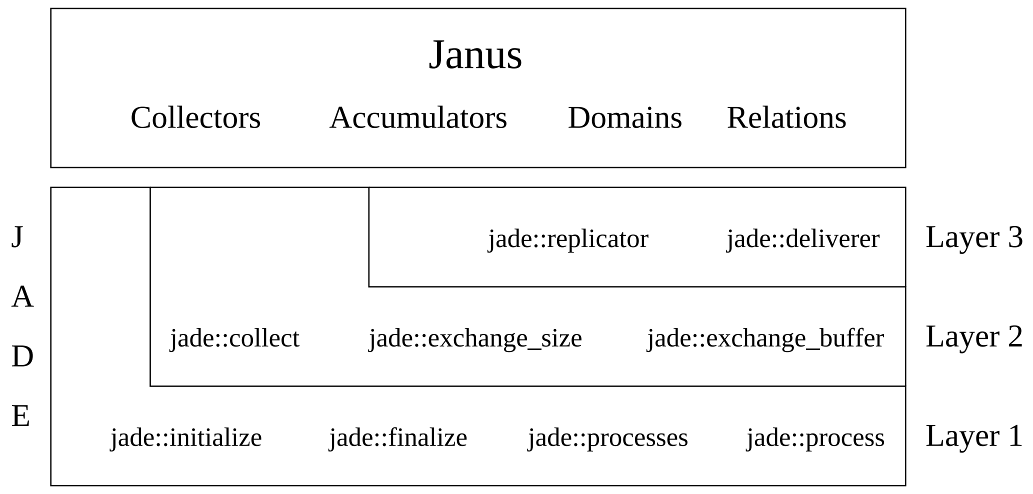

Figure 6.2: Layers of Jade

The third layer is constituted by the two template classes `jade::deliverer` and `jade::replicator`. These classes transfer data in a two-phase manner, that is, elements are first inserted and buffered (if necessary) until they are send. Among other methods, both classes have a `freeze` method that is an collective operation.

6.3.1 JADE Layer 1 Components

The layer 1 of JADE is formed by counterparts of the Janus runtime environment functions of Section 5.6.1.

```
void    jade::initialize(int& argc, char**& argv);
void    jade::finalize();
size_t  jade::processes();
size_t  jade::process();
```

An MPI implementation of this functions is given in Section 6.3.4 on Page 112.

6.3.2 JADE Layer 2 Components

6.3.2.1 The `collect` Function

The template function `jade::collect` has two arguments.

```
    template<class T>
    void collect(const T& t, std::vector<T>& result);
```

The first argument `t` with which `collect` is called in the i-the process becomes the i-th entry of the vector `result`. The length of the result vector equals the number of parallel processes.

6.3.2.2 The exchange_size Function

The function `exchange_size` has two arguments of type `jns::size_vector` whose length must equal the number of parallel processes.

```
    void exchange_size(const size_vector& input,
                             size_vector& output);
```

The k-th element of the input vector of the i-th process is copied into the i-th element of the output vector of the output vector.

6.3.2.3 The exchange_buffer Function

The template function `exchange_buffer` has six arguments which come in two groups of three arguments. The first three arguments describe the data that shall be sent whereas the second group describes the data to be received by this collective operation.

```
  template<typename T>
  void exchange_buffer(const T*            send,
                       const size_vector&  size1,
                       const size_vector&  offset1,
                             T*            recv,
                       const size_vector&  size2,
                       const size_vector&  offset2);
```

Each group of argument has the same structure.

1. A pointer of type `T` that holds the starting address of the data to be sent or received, respectively.
2. A `jns::size_vector` object whose length equals the number p of parallel processes. For i-th process the k-th element of this array describes the *number* of consecutive elements in the buffer that shall be (sent to)/(received from) k-th process.

3. A `jns::size_vector` object whose length equals $p+1$. For i-th process the k-th element of this array is the *offset* in the buffer at which the data to/from the k-th process are sent/received.

6.3.3 Jade Layer 3 Components

6.3.3.1 The `deliverer` Template Class

The template `jade::deliverer` in Figure 6.3 collects objects and transfers them to their designated location. Collection and transfer happens in a two-phase way.

```
template<typename T>
class deliverer {
public:
  typedef T value_type;

  void insert_at(const value_type& v, size_t p);

  void freeze();

  size_t size() const;

  const value_type&  operator[](size_t) const;
};
```

Figure 6.3: The `deliver` template class

In the first phase, elements are inserted into the deliverer object using the `insert_at` method. The second argument of this method denotes the process to which the element shall be delivered. After calling the `freeze` method, the second phase starts. Note that `freeze` is an collective operation that must be called in all parallel processes. In the second phase, all inserted elements have been delivered to their respective processes. The method `size` returns the number of locally accessible objects and can only be called after the deliverer has been frozen. Delivered objects can be accessed with `operator`.

The semantics of the operations of this class is very similar to those of Two-Phase Domain (§4.2.3) and the collectors of Section 5.5.1. In fact, the class `deliverer` is used for the implementation of all collectors—except the `conversion_collector` for which the template class `replicator` from Section 6.3.3.2 is used.

The implementation of `deliver` is quite simple. In the first phase, inserted elements are stored into a temporary buffer and prepared to be sorted with respect to their process. Within `freeze` the following steps are performed.

1. Elements are sorted with respect to their designation process. Hereby the *counting sort* algorithm mentioned in Section 6.2 is deployed. Counting sort can be used here because the keys are from a bounded set of integers.

2. After sorting the input buffer, the elements that have to be sent a process are *consecutively* stored. The numbers of elements to be sent and their offsets within the buffer are determined and exchanged using the `jade::exchange_size` function (§6.3.2.2).

3. After receiving the size information of data to be sent, the size of the receive buffer and the offsets for the data from the different processes can be determined.

4. Finally, the inserted and sorted data are exchanged using the `jade::exchange_buffer` function (§6.3.2.3).

Thus, communication that occur in this class is completely expressed by using JADE functions of layer 1.

6.3.3.2 The `replicator` Template Class

The interface of the `replicator` template in Figure 6.4 is very similar to that of the class `deliverer` (Figure 6.3.3.1 The only difference is that there is only a method `insert(const value_type&)` which does *not* allow to specify mapping information. The semantic of the insert, freeze, and access methods of this class is a follows.

The elements that have been inserted in the process p can, after freezing it, be accessed at all *other* processes. The methods `size` and `operator[]` are the means for access. As in the case of `replicator`, all communication operations are implemented using the `exchange_size` and `exchange_buffer` methods of JADE.

Thus this class *replicates* inserted elements on other processes. This feature is very useful when objects for which no mapping information is available

```
template<typename T>
class replicator {
public:
  typedef T value_type;

  void insert(const value_type& v);

  void freeze();

  size_t size() const;

  const value_type&  operator[](size_t) const;
};
```

Figure 6.4: The `replicator` template class

must be globally evaluated. However, clients of this class must bear in mind the high amount of memory that is consumed to store many replicated objects.

The class `replicator` is used to implement the template `conversion_collector` (§5.5.1.3) that transforms pairs of domain element and associated data objects into local positions and associated data objects.

- Pairs inserted in this collector are checked whether there first components is an element of the local subdomain.
  - If the first component is locally contained then it can be immediately converted into its position.
  - If it cannot be locally resolved then the pair is inserted in to the replicator.
- The freeze method of `conversion_collector` calls the freeze method of the replicator. Thus unresolved pairs can now be checked in all other processes and eventually resolved.

6.3.4 Porting Jade to MPI

Message-passing has become accepted as a portable style of parallel programming on distributed memory architectures.

The Message Passing Interface (MPI) standard is a library specification for message-passing[97]. It was designed for high performance on both massively parallel machines and on workstation clusters. MPI was primarily designed to enable the development of *parallel* libraries. Among the MPI abstractions that simplify writing parallel MPI libraries are *process groups* that describe participants of collective operations and *topologies* that describe process relationships.

This section describes a simple MPI[97] implementation of the layer 1 and layer 2 functions of JADE. Table 6.1 shows which MPI functions have been mainly used for the implementation of the JADE/Janus constructs.

JADE	Section	Implemented with MPI function
`jade::initialize`	§6.3.1	`MPI_Init`
`jade::finalize`	§6.3.1	`MPI_Finalize`
`jade::processes`	§6.3.1	`MPI_Comm_size`
`jade::process`	§6.3.1	`MPI_Comm_rank`
`jade::gather`	§6.3.2.1	`MPI_Allgather`
`jade::exchange_size`	§6.3.2.2	`MPI_Alltoall`
`jade::exchange_buffer`	§6.3.2.3	`MPI_Alltoallv`

Table 6.1: Mapping of JADE to MPI functions

This table shows a close correspondence of JADE and MPI functions. For communications routines, that is for routines from layer 2, *collective* MPI functions have been used.

A major difference to MPI is that the routines `gather` and `exchange_buffer` are template functions. MPI functions can also be called with various types but the type the function is used for must be explicitly specified by arguments. This is contrast to a C++ template function that constitute a family of overloaded functions.

In order to use the MPI functions for the implementation of the generic data movements of JADE Layer 2 the following condition must be fulfilled by the template argument `T`.

- The bytes constituting an object of type `T` are contiguous.
- The bytes constituting an object of type `T` can be copied with `std::memcpy`
  - to a sufficiently large array of `char` and back again without changing the object's value,

    – to another object of the type `T`, in which case the second object's value will be the same as that of the first.

Examples of types that fulfill these requirements are `int`, `float`, `std::pair<short,double>`, or `boost::array<int,3>` which covers most of the types used in the examples of Chapter 7 and inside the components of Janus.

More generally said, these conditions are fulfilled by so-called *plain old data* (POD)—a category of types specified by the C++ standard [101] which includes

- Scalar types, i.e.,
    – arithmetic types,
    – enumeration types,
    – pointer types, and
    – pointer-to-member types.
- POD class types, which include classes *without* any of the following members
    – non-static data (including arrays) of any pointer-to-member type,
    – non-static data (including arrays) of any non-POD class type,
    – non-static data of any reference type,
    – user-defined copy assignment operator, nor
    – user-defined destructor.

6.4 Support for Shared Memory Systems

Since there are implementations of MPI for shared-memory systems, the MPI port of JADE can run there as well. However, the message passing paradigm is neither a natural nor efficient choice on shared-memory systems.

The OPENMP Application Program Interface[102] supports multi-platform shared-memory parallel programming in C/C++ and Fortran. It is a relatively new effort to boost shared memory programming on Unix platforms and Windows NT platforms. OPENMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications.

OpenMP specifies compiler directives, library routines, and environment variables to express shared-memory parallelism. OpenMP fully supports loop-level parallelism and to some extend permits nested parallelism. General task parallelism is not supported by OpenMP.

In the paper[42], it has been described how OpenMP can be integrated into Janus. As indicated in Figure 6.1, local components of Janus have been enhanced by OpenMP directives. Hereby, mostly compiler directives that annotate loops have been used. For simple applications such as the cellular automaton simulation *Game of Life* (see Section 7.1), good speedup result could be reported.

However, a problem is that in more complex applications much more loops occur *outside* of Janus. With parallel processes, that is on top of Jade, these loops are implicitly parallelized whereas with OpenMP they must be explicitly annotated to perform in parallel.

Moreover, OpenMP requires an OpenMP-aware *compiler* and only a few are available, for examples KAI's Guide compiler[66]. This is in strong contrast to MPI, which is a *library* specification that does not require additional tool support. Also, there are many MPI implementations including free ones available[48].

For these reasons, the support for Janus on shared-memory architectures is not yet as mature as on distributed-memory architectures.

6.5 Conclusions

In this chapter it has been shown how the components of Janus can be configured for different parallel platforms. Jade, a parallel port package for Janus, has been presented. Jade provides several layers of abstraction to hide details of an underlying distributed-memory architecture. The components of Jade are mostly (template) functions that can be easily ported to MPI or other message passing systems. However, Jade provides also template classes that implement higher level data movement operations that are particular useful for the implementation of Janus' two-phase data structures and algorithms.

The relation of Janus two shared memory platforms has also been investigated. Janus can exploit the parallel processing power of these platforms but not as transparently as for distributed-memory platforms.

Chapter 7

Application Engineering with Janus

The Janus conceptual framework and its generic components of Chapter 4 and Chapter 5 are the reusable work products for the domain of data parallel applications. Their design rest on the domain analysis that was performed in Chapter 3.

As pointed out in Section 2.3, the work products of *domain engineering* are reused during *application engineering*, that is, when engineering concrete systems in the domain.

In this chapter, the impact and the suitability of Janus on the implementation of broad field of data parallel application is investigated. This is done by implementing representative data-parallel applications The chosen applications are closely related to those considered in Chapter 3. In particular the following problems are considered:

- Conway's *Game of Life* (see Section 7.1) which is cellular automaton simulation that is performed on a two-dimensional grid and involves stencil-like communication patterns. It is shown how simple application-oriented communication patterns can be expressed with Janus.

- *Assemblage and solution* of finite element method—see Section 7.2. Both tasks constitute key components of finite element methods and require Janus components that support irregular sparse structures. As an example, a simple two-dimensional finite element method is considered and it is explained how the concrete, application-dependent data structures can be instantiated from generic Janus components. The efficient assemblage of sparse matrices uses one of Janus' two-phase algorithms. The assembled linear system is solved with an iterative method that uses the `pull_matrix` communication method of relation classes.

- *Red/Green* refinement of triangulations is considered in Section 7.3. This example extends the data structures used in Section 7.2. The emphasis of this section is on the use of Janus communication operations that allow to evaluate individual (remote) elements.

- *Bellman-Ford* single-source shortest path algorithm—see Section 7.4. This example shall illustrate the suitability of Janus for data-parallel graph algorithms. Similar to the example on mesh refinement (Section 7.3) the emphasis is on using Janus communication operations for accessing individual elements.

Finally, a few general principles are formulated that are helpful when writing data-parallel applications within the Janus framework. Performance results will be presented in Chapter 8.

7.1 Conway's Game of Life

A mathematical description of this cellular automaton simulation has been given in §3.1.2. As pointed out there, *Conway's Game of Life* rests on a two-dimensional rectangular grid and uses an eight-point stencil to compute the new state in each grid point.

In this section it is shown how the Janus components `grid` and `stencil` can be deployed to provide a parallel implementation. In particular it is shown how two-dimensional grids are declared and how an application-dependent stencil can be described with Janus.

7.1.1 Grid Declaration

The grid that is at the bottom of Conway's *Game of Life* can be defined with the `grid` template §5.2.2.1.

```
jns::grid<2> g(lower, upper);
```

The template parameter of `grid` specifies the grid dimension. The arguments `lower` and `upper` of the grid object `g` are of type `jns::grid<2>::value_type`. In order to describe the rectangular grid $[0, m) \times [0, n) \subset \mathbb{Z}$, one can initialize `lower` and `upper` in the following way:

```
lower[0] = 0; lower[1] = 0;

upper[0] = m; upper[1] = n;
```

7.1.2 Definition of the Eight-Point Stencil

A model of Relation that efficiently represents an eight-point stencil of Equation 3.7 can now be easily built by using the `jns::stencil` template (§5.3.2.1).

```
jns::stencil<2,8>
```

Note that the first template parameter of `stencil` is the dimension of the underlying grid. The second parameter is the *length* of the stencil which is in this case 8.

The offsets of the eight-point stencil are return by a short inline function `life_stencil`:

```
boost::array<boost::array<int,2>,8>
life_stencil()
{
   boost::array<boost::array<int,2>,8>  r = {{
         {{ -1, -1 }}, {{ -1,  0 }}, {{ -1,  1 }},
         {{  0, -1 }},               {{  0,  1 }},
         {{  1, -1 }}, {{  1,  0 }}, {{  1,  1 }} }};
   return r;
}
```

The pairs of braces `{{` and `}}` are an artefact of the implementation of `boost::array` as an aggregate[28, 99].

In order to define the concrete stencil object it must be specified on which subset of the particular grid the stencil can be correctly applied. This is necessary in order to avoid the application of a stencil on the boundary of the grid where not all neighbors might exist.

In the case of the eight-point stencil of Equation 3.7 the stencil can be safely applied on subgrid $[1, m-1) \times [1, n-1)$ of $[0, m) \times [0, n)$.

For this two new temporary variables lower1 and upper1 of type jns::grid<2>::value_type are declared

```
lower1 = {{ lower[0]+1, lower[1]+1 }};
upper1 = {{ upper[0]-1, upper[1]-1 }};
```

and initialized with the help of the previously defined (§7.1.1) grid bounds lower and upper. Hereby, the fact that boost::array is implemented as an C++ *aggregate* is exploited[99].

Finally, the definition of the concrete eight-point stencil object s on the grid g reads:

```
jns::stencil<2,8> s(g, life_stencil(), lower1, upper1);
```

7.1.3 Application of the Eight-Point Stencil

The state (alive or dead) of the cells on the the grid object g can be easily represented by the STL container std::vector<int> which is a model of Property Function(§4.3). The declaration of a property function object on g reads:

```
std::vector<int> x(g.size());
```

The stencil object s defined in §7.1.2 will be used to compute the sum of Equation 3.9 of living neighbor cells. When living cells are represented by 1 and dead cells by 0 the sum is, as stated in §3.1.2 the number of living neighbors.

The template member pull_reduce (§4.4.5.1) of stencil will be used to compute these sums for all points of the grid g. The application-specific visitor object LifeV uses these sums and the state of the cells to compute the next generation.

The class LifeV in Figure 7.1 implements the rules of Equation 3.10 to compute for a grid point the state of a cell in the *next* generation. The class LifeV satisfies the requirements of a relation visitor (Table 4.19), that is, it provides

```
void operator()(size_t i, int sum)
```

where `i` is the *position* of the grid point under consideration and `sum` the number of living neighbors.

```
struct LifeV {
    const int* a;
          int* b;

    LifeV(const int* a_, int* b_) : a(a_), b(b_) {}

    void operator()(size_t i, int sum) {
        b[i] = (sum == 3) ? 1 : (sum == 2) && a[i];
    }
};
```

Figure 7.1: *Evaluate* visitor for Life

With these preparations a short Life simulation on the grid `g` (consisting of 100 iterations) finally reads:

```
std::vector<int> y(g.size());
for(size_t i = 0; i < 100; i++) {
   s.pull_reduce(&x[0], LifeV(&x[0], &y[0]));
   x.swap(y);
}
```

The property function `y` serves as a temporary variable that holds the new state. By using the `swap` method of `std::vector` the property function `x` is reinitialized with `y` at the end of each simulation step.

A temporary `LifeV` visitor object that is initialized with the starting addresses of the state functions `x` and `y` is passed as second parameter to the `pull_reduce` method (§4.4.5.1). The first argument of `pull_reduce` is also the address to the data stored by `x`.

7.1.4 Discussion

In this section it has been shown how *Conway's Game of Life* can be implemented using the Janus components `grid` and `stencil`. The core of this simulation is the repeated evaluation of an eight-point stencil on a static tow-dimensional grid.

A parallel implementation for a distributed-memory platform has to deal with fact that some of the neighboring grid points are located on different processors. For clients of Janus, however, this aspect is only of little concern. After declaring the stencil and instantiating it for a particular grid, all data movements that are caused by the application of the stencil are automatically handled by the `pull_reduce` method required by Relation. Moreover, the data movement and subsequent computation can be easily combined since `pull_reduce` can be customized through a *Visitor* class.

The expressiveness of the Janus implementation is comparable with the parallel implementation of *Life* that uses p3l —a structured parallel language based on skeletons[27]. Skeletons are a fixed set of patterns that provide a *restricted* parallel computation model.

The performance of the Janus implementation of *Life* is investigated in Section 8.2 on Page 165.

7.2 A Simple Finite Element Method

In this section, it is outlined how a simple parallel finite element method can be implemented using the Janus components of Section 5.

In accordance with Hackbusch[52], the Poisson problem

$$-\Delta u(x) = f(x) \qquad x \in \Omega \tag{7.1}$$

$$u = 0 \qquad \text{on } \partial\Omega \tag{7.2}$$

with homogeneous boundary conditions (see Equation 7.2) on a polygonal domain is chosen as model boundary value problem. For sake of simplicity *linear triangle elements*[19] are applied to provide a discretization of this problem. This simple setting requires that only the triangles and vertices of the triangulation have to be represented (see Section 7.2.1). It is explained how a distributed representation of these sets and of the triangle-vertex relation can be easily described.

The key points of this section is the use of the Janus classes `domain` and `relation` in order to represent the components of the irregular mesh and its relations. Section 7.2.1 also discusses the initialization of user-defined domain types and associated values.

In Section 7.2.2 , it is shown how the discretized differential operator is *assembled* from so-called element matrices that are defined on the triangles. It is also outlined how, using the triangle-vertex relation, the element matrices are computed. For the efficient assemblage of the element matrices the Janus algorithm `relation_accessor` is used.

Finally, in Section 7.2.3, it is shown how the *conjugate gradient iterative method* for the solution of the discrete system can be implemented in Janus. The emphasis of this section is on the use of the `pull_matrix` method of Relation and of parallel versions of STL algorithms to express the core components of an iterative method.

7.2.1 Sets and Relations

In this section it is explained how the different sets and relations that form a triangulation are represented using domain and relation classes of Janus. Figure 3.5 on Page 44 shows the sets of triangles T and set of vertices V of the triangles. The emphasis is on the initialization of concrete two-phase data structures and the application-dependent attributes that are associated with them.

7.2.1.1 Defining Domain and Relation Types

Vertices are represented as integers whereas triangles are described as the triple (a, b, c) of its three vertices a, b, and c. The corresponding type definitions read:

```
typedef  int                    Vertex;
typedef  boost::array<Vertex,3>  Triangle;
```

Hereby the fixed-size `array` template of the Boost library[28] is used.

With these types corresponding models of domains can be easily defined using the `sorted_set` template (see §5.2.1.1).

```
typedef  jns::sorted_set<Vertex>    Vertices;
typedef  jns::sorted_set<Triangle>  Triangles;
```

Relations of domain elements must be explicitly defined in Janus. This also holds for such an “obvious” relation such as the one that associates the vertices with a triangle. Note that this relation is implicitly given through the way triangles are described.

Since the triangle-vertex relation is a *sparse* relation it can be naturally represented using the `relation` template of Janus.

```
typedef  jns::relation    Relation;
```

7.2.1.2 Associating Properties with Domains

A mesh generator such as TRIANGLE[93] provides not only the lists of vertices and triangles of a triangulation but also certain attributes that are associated with these geometric objects. Examples of such attributes are the Cartesian coordinates of a vertex and flag that indicates whether a vertex is on the boundary of the domain or not.

There are two major ways to represent such attributes. The first possibility is to *augment* the `Vertex` data type so that it contains the attributes as subobjects. The following code fragment declares a type `AugmentedVertex` that contains subobjects of type `Vertex` and `bool` to represent a boundary flag.

```
struct AugmentedVertex {
    Vertex  vertex;
    bool    flag;
};
```

The close coupling of primary data (in this case the vertex) and attributes (here the boolean flag) has obvious advantages. However, from the point of view of high performance this close coupling has also drawbacks since it easily leads to "heavy-weight" objects that in contrast to type `Vertex` cannot be held in registers[1] and lead to more frequent cache misses. Another drawback of this approach is that a client of Vertex had to modify her/his augmented data structure whenever new attributes must be added.

The second approach rests on associating one or more ranges of random access iterators whose value type represents the attribute(s) with a domain. This is the basic idea of property functions of Section 4.3 and it is also applied in this case. Therefore at first the following attribute types are introduced:

```
typedef  short                    Flag;
```

[1]The ability to keep objects on the stack and even to hold them in registers is a key performance advantage of C++ over other languages, for example Java. For this reason, the newer C# programming language introduces *value types* (in contrast to *reference types*) to represent light-weight objects

```
typedef  double                Real;
typedef  boost::array<Real,2>  Coordinate;
```

Then the `vector` class from the C++ standard library is deployed to define types that can describe ranges of appropriate random access iterators.

```
typedef  std::vector<Flag>        Flags;
typedef  std::vector<Real>        Reals;
typedef  std::vector<Coordinate>  Coordinates;
```

In the following section it is explained how domain objects and associated property functions can be initialized.

7.2.1.3 Initialization of Domains and Properties

At first the initialization of the triangle set is considered. The case of triangles is easy since in the setting of linear triangle elements there are no additional attributes associated with a triangle—except the mapping information of the triangle. These data are contained in a file where each line consists of four integers

$$a \quad b \quad c \quad m.$$

The first three integers represent a triangle whereas the last integer m is the process number to which the triangle (a, b, c) shall be mapped.

The mapping information can be provided by applying partitioning tools such as METIS[63] or MOSAIK[103]. More important, however, is to understand that Janus does not provide any mapping procedures (except for the `grid` template of §5.2.2.1). This important concern of parallel computing is deliberately left to tools like METIS or MOSAIK.

Reading the triangles from a file and distributing them according to the mapping specified there is shown in Figure 7.2.

Two remarks are important for a better understanding of the code in Figure 7.2.

1. Since every element may be inserted only once into a domain (cf. Table 4.5 it is necessary to perform the read and insert operations only on one process (in this case process 0). The `freeze` method, however, must be called in all participating processes.

2. The distribution of the triangles is only completed when the `freeze` method of the `Triangles` class has been called. The `insert_at` method only *indicates* the mapping. This has been explained in Section 4.2.3.4.

```
void read(fstream& file, Triangles& triangles)
{
    if (jns::process() == 0)
    {
        Triangle t;
        size_t   p;
        while((file >> t) &&  (file >> p))
            triangles.insert_at(triangle, p);
    }
    triangles.freeze();
}
```

Figure 7.2: Reading and distribution of triangles

Figure 7.3 shows a partitioning of the triangulation of Figure 3.5 onto seven processes. This triangulation was created using the TRIANGLE[93] mesh generator. As partitioner METIS[63] was used.

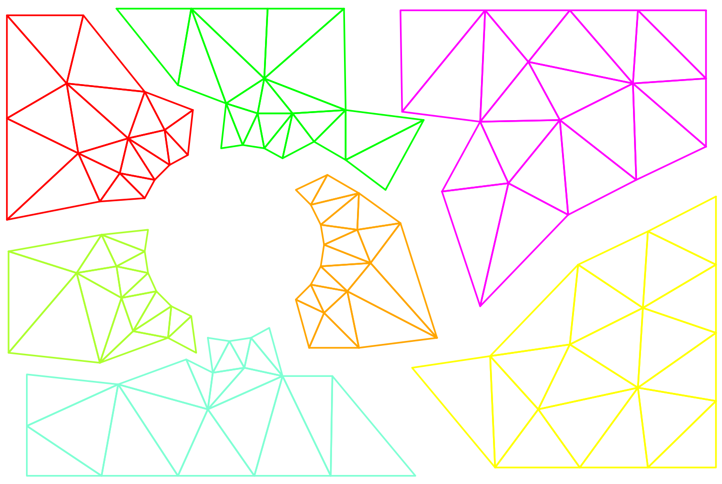

Figure 7.3: Partitioning of a triangulation

A bit more complicated is the initialization of vertex set of the triangulation because there are two attributes associated with a vertex. These are, on the one hand, the Cartesian coordinates which will be represented as objects of type `Coordinate`. On the other hand, there is a boolean flag associated with each vertex where *true* indicates that the vertex is on the boundary of the domain. Boolean and other flags will be represented as objects of type `Flag`.

For sake of clarity, only one vertex attribute, namely the Cartesian coordinates are considered. Moreover, as in the case of triangles, a line-oriented input format is assumed. This means that each line of the input file has the structure

$$v \quad x \quad y \quad m$$

where v denotes the vertex, (x, y) its Cartesian coordinates, and m the process number to which the vertex shall be mapped.

The following Figure 7.4 shows a function that implements the task of reading the vertex input and distributing it onto the parallel machine.

The major differences to reading and distributing the triangles (see Figure 7.2) are explained here.

1. A *domain collector* is introduced that distributes vertices and associated attributes according to the specified mapping info. The template class `collector<A,B>` handles objects of type `std::pair<A,B>`, that is in this particular case `std::pair<Vertex,Coordinate>`. For more details see Section 5.5.1.1.

2. After *freezing* the collector all vertices and their respective attributes have been transferred to the specified processes. As a consequence the `insert` method instead of `insert_at` can be used to insert the vertices that have been retrieved from the collector.

3. After freezing the vertex domain, the number of locally stored vertices is known. This number is used to give the property function that shall hold the vertex coordinates the correct size. Then again the collected data are retrieved from the collector. For each retrieved object of type `std::pair<Vertex,Coordinate>` the position of first component (i.e. the vertex) is searched in the `vertices` domain and the second component (i.e. the Cartesian coordinates) is assigned at that position of the property function `c`.

Note that this procedure can be easily extended to the case of several attributes by providing an appropriate class as second template parameter of `collector`.

```
void read(fstream& file, Vertices& vertices, Coordinates& c)
{
    jns::collector<Vertex,Coordinate> collector;
    if (jns::process() == 0)
    {
        Vertex     v;
        Coordinate xy;
        size_t     m;
        while(file >> v && file >> xy && file >> m)
            collector.insert(v, xy, m);
    }
    collector.freeze();

    for(size_t i = 0; i < collector.size(); ++i)
       vertices.insert(collector[i].first);
    vertices.freeze();

    c.resize(vertices.size());
    for(size_t i = 0; i < collector.size(); ++i) {
       std::pair<Vertex,Coordinate> a = collector[i];
       size_t pos = vertices.position(a.first);
       c[pos] = a.second;
    }
}
```

Figure 7.4: Reading and distribution of vertices and their coordinates

7.2.1.4 Initialization of the Triangle-Vertex Relation

The triangle-vertex relation plays a special role when computing the discretization of the Laplace operator in Equation 7.1. As it name suggests, this relation describes for each triangle of the triangulation what are the (global) positions of its three vertices—see Figure 7.5.

The triangle-vertex relation is used to *access* the coordinates of the vertices *from* the triangles.

After initializing the triangle set `triangles` and the vertex set `vertices`, this relation can be easily defined. Hereby, the Janus algorithm `n_relation_insert` of Figure 5.8 is deployed.

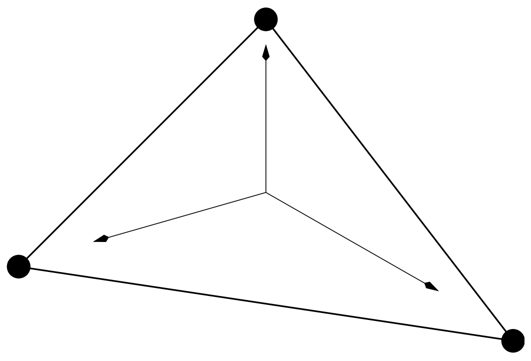

Figure 7.5: Triangle-vertex relation

First a function object that describes the relationship between a triangle and its vertices must be defined. The definition is trivial because the triple of vertices that belong to a triangle constitutes this triangle.

```
struct TriangleVertices {
    typedef  Triangle               argument_type;
    typedef  boost::array<Vertex,3>  result_type;

    result_type operator()(argument_type a) const
    {
         return a;
    }
};
```

With this function object the code for the initialization of the triangle-vertex relation object `tv` reads:

```
Relation tv(triangles, vertices);
TriangleVertices gen;
jns::n_relation_insert(triangles, vertices, gen, tv);
tv.freeze();
```

Figure 7.6: Initialization of the triangle-vertex relation

Thus, the generic Janus algorithms can be easily customized to initialize an application-dependent relation. The computations and necessary data-movement operations are hidden from the user.

7.2.2 Sparse Matrix Assemblage

The discretized Laplace operator (Equation 7.1) is computed by *assembling element matrices* according to Equation 3.12 on the triangles of triangulation. For linear triangle elements, there are three degrees of freedom associated with each triangle. These degrees of freedom are located at the vertices of the triangle. Note that Equation 3.12 can be simplified by using the Term 3.13. In order to compute this term, the coordinates of the vertices of all triangles must be evaluated on the triangles. Therefore the triangle-vertex relation of Figure 7.6 plays an important role in the code of Figure 7.7.

Two versions of sparse matrix assemblage are considered. In the first version, both the sparse relation and the matrix coefficients associated to it are assembled. In the second version, it is assumed that the sparse relation has already been constructed and only the coefficients have to be computed. The second version is more efficient when several matrices with the same sparsity pattern have to be computed. This frequently occurs when solving time dependent problems.

7.2.2.1 Assembling Relation and Coefficients

The structure of the code in Figure 7.7 is similar to that of reading the vertices and their coordinates in Figure 7.4.

1. Objects that consist of a *key* component and a *data* component are distributed according to the mapping of their key using a collector object.

2. The spatial structure (in this case the relation) is initialized using the collected key components.

3. Finally, the data components of the collected objects are used to initialize a *property function* that holds additional attributes of the spatial structure.

A notable difference to the reading of Vertices in Figure 7.4, however, is that in *all* processes the data are inserted in the collector object. Here follows a more detailed description of the code in Figure 7.7.

1. (a) The triangle-vertex relation `tv` is used (as mentioned above) to access the coordinates of the vertices of the triangles. Therefore

```
void assemble(const Vertices& vertices, const Relation& tv,
              const Coordinates& coordinates,
              Relation& r,   Reals& coefficients)
{
  Relation::accessor<Coordinate> accessor(tv, coordinates);
  accessor.pull();
  jns::relation_collector<Real> coll(vertices, vertices);
  for(size_t i = 0; i < tv.size1(); ++i) {
      std::vector<Coordinate> value(tv.size(i));
      std::vector<size_t>     index(tv.size(i));
      for(size_t k = 0; k < tv.size(i); ++k) {
         value[k] = accessor.get(i,k);
         index[k] = tv.second(i,k);
      }
      Matrix33 e = laplace_linear_element_matrix(value);
      for(size_t m = 0; m < index.size(); ++m)
        for(size_t n = 0; n < index.size(); ++n)
          coll.insert(index[m], index[n], e(m,n));
  }
  coll.freeze();

  for(size_t i = 0; i < coll.size(); ++i)
      r.insert(coll[i].first);
  r.freeze();

  coefficients.resize(r.size());
  for(size_t i = 0; i < coll.size(); ++i) {
     std::pair<jns::pair_size_type,Real> p = coll[i];
     coefficients[r.position(p.first)] += p.second;
  }
}
```

Figure 7.7: Assemblage of the sparsity pattern and coefficients

at first, an appropriate *accessor* object is defined and updated by calling its *pull* method (see Section 4.4.4).

(b) A `relation_collector` (Section 5.5.1.2) is declared and bound to the vertices of the triangulation. This collector objects will temporarily hold and deliver pairs of relation elements and associated real numbers to their respective processes.

   (c) For all triangles (locally this number equals `tv.size1()`), two auxiliary arrays `value` and `index` are initialized with the coordinates and the (global) positions of the vertices. Hereby the methods `second` of Table 4.15 and `get` of Table 4.18 are used.

   (d) The array `value` whose size equals the number `tv.size(i)` of vertices that belong to the i-th triangle (that is 3) is passed to the function `laplace_linear_element_matrix` which computes the corresponding 3×3 element matrix object `e` of type `Matrix33`.

   (e) The 9 matrix coefficients of the element matrix `e` are inserted together with their indices in the collector. After all element matrices have been computed the collect is frozen.

2. After freezing the collector, the pairs of indices and contributions to the matrix coefficients are stored in the collector. At first the key components, that is the relation elements are inserted into the relation object `r` and the relation is frozen, too.

   (a) Note that this relation `r` constitutes the *nearest-neighbor relation* of the vertices of the mesh that is two vertices belong to `r` if there is a triangle which both belong.

   (b) Note also that most of the index pairs are inserted twice into the relation. This is due to the fact that two triangles that share the vertices with positions x and y compute a contribution to the index pair (x, y). Multiple inserted index pairs are removed and only one copy of them left in the relation during the execution of the `freeze` method.

3. Finally, the collector is traversed again and the contributions to the matrix are added to the matrix coefficients `m` at the position of the corresponding index pair in the relation `r`.

7.2.2.2 Assembling only Coefficients

The Janus implementations of this version of matrix assemblage (shown in Figure 7.8) is a simplified version of the general sparse matrix assemblage routine of Figure 7.7.

Major differences are that instead of `relation_collector` an object of type `relation_accumulator` (§5.5.2.3) is used. The relation accumulator `accu` is initialized with the assembled relation `r`. The accumulator adds the inserted values to the according positions of the `coefficients` property

```
void assemble(const Vertices& vertices, const Relation& tv,
              const Coordinates& coordinates,
              const Relation& r,   Reals& coefficients)
{
  Relation::accessor<Coordinate> accessor(tv, coordinates);
  accessor.pull();
  jns::relation_accumulator<Relation,Real> accu(r,coefficients);
  for(size_t i = 0; i < tv.size1(); ++i) {
      std::vector<Coordinate> value(tv.size(i));
      std::vector<size_t>     index(tv.size(i));
      for(size_t k = 0; k < tv.size(i); ++k) {
         value[k] = accessor.get(i,k);
         index[k] = tv.second(i,k);
      }
      Matrix33 e = laplace_linear_element_matrix(value);
      for(size_t m = 0; m < index.size(); ++m)
        for(size_t n = 0; n < index.size(); ++n)
          accu.insert(index[m], index[n], e(m,n));
  }
  accu.freeze();
}
```

Figure 7.8: Assemblage of Coefficients only

function. Note that these operations can be delayed until the `freeze` method has been called.

The `Relation` argument `r` is passed as non-mutating reference—indicating that the relation has already been assembled.

7.2.2.3 Discussion

Assemblage of element matrices is a key component of finite element programs. This section presented two Janus implementations. The first version assembles both the sparse relation and the associated matric coefficients. The second versions assumes that the sparse relation has already been assembled and only coefficients have to be accumulated. Both scenarios can occur in a finite element program and it is therefore important that Janus can support them.

7.2.3 Implementation of the Conjugate Gradient Method

Figure 7.9 shows the parallel implementation of the methods of *conjugate gradient* within the Janus framework. This implementation rests on the conjugate gradient method that is part of the *Iterative Matrix Library* (IML++)[30]. Major differences are that overloaded numerical operators are *not* deployed in the Janus implementation. Rather the `pull_matrix` method and simple template functions are used to express basic linear algebra operations. This approach is similar to that of *Iterative Template Library*[71].

7.2.3.1 Template Parameters and Signature

The function `cg` in Figure 7.9 has the following template parameters.

- `Rel` which is a model of Relation and represents the sparse matrix format of the linear system to be solved.
- `Vec` is also a model of Property Function that represents right hand side and solution of the liner system to be solved and moreover the coefficients of the matrix.
- `Prec` represents the preconditioner used in the conjugate gradient method.
- `Real` shall be the value type of `Vec`.

The arguments of the function `cg` are:

- the relation object `R` and the associated matrix coefficients `m`,
- the solution vector `x` and the right hand side `b` of the equation,
- the preconditioner `prec`, and
- the maximal number of iterations (`max_iter`) to be performed and the value `tol` that indicates when the conjugate gradient method can be terminated.

7.2.3.2 Implementation

The source code in Figure 7.9 emphasizes the four major algorithmic components already mentioned in Section 3.1.3.3.

```
  template<class Rel, class Vec, class Prec, class Real>
  std::pair<size_t, Real>
  cg(const Rel& R, const Vec& m, Vec& x, const Vec& b,
     const Prec& prec, std::pair<size_t,Real> control)
  {
    Real normb = norm(b); if (normb == 0.0) normb = 1;
    Vec r(x.size());
    R.pull_matrix(&m[0],&x[0],jns::assign_visitor<Real>(&r[0]));
    scale_add(r, -1.0, b);

    Vec p(x.size()), q(x.size());
    Real rho, res = 0;
    for (size_t i = 1; i <= control.first; i++) {
      prec.solve(r,q);
      Real rho_old = rho;
      rho = dot(r,q);
      scale_add(p, (i == 1) ? 0 :  rho / rho_old, q);

      R.pull_matrix(&m[0],&p[0],jns::assign_visitor<Real>(&q[0]));
      Real alpha = rho / dot(p,q);
      update(x,  alpha, p);
      update(r, -alpha, q);

      res = norm(r)/normb;
      if (res <= control.second) return std::make_pair(i,res);
  }
  return std::make_pair(control.first,res);
}
```

Figure 7.9: Implementation of the conjugate gradient method in Janus
.

Matrix-vector multiplication is expressed using the `pull_matrix` method on the relation object `R` together with the coefficients `m`. Note that the `pull_matrix` method handles any communication that might result from a distributed representation of the matrix.

Vector updates are expressed using the template functions `scale_add` and `update` whose simple implementations are shown here.

```
    template<typename Vector, typename Scalar>
```

```
void scale_add(Vector& w, Scalar s, const Vector& v) {
    for(size_t i = 0; i < w.size(); ++i)
        w[i] = v[i] + s*w[i];
}

template<typename Vector, typename Scalar>
void update(Vector& w, Scalar s, const Vector& v) {
    for(size_t i = 0; i < w.size(); ++i)
        w[i] += s*v[i];
}
```

Both are communication-free methods.

Inner products are represented by the functions `dot` and `norm`. Their implementation rests on the parallel version (see Section 5.5.3) of the STL algorithm `inner_product`.

```
template<typename Vector>
typename Vector::value_type
dot(const Vector& v, const Vector& w)
{
  typedef typename Vector::value_type V;
  return jns::inner_product(v.begin(), v.end(), w.begin(), V());
}
```

Preconditioning is the most complicated part of the conjugate gradient method. Numerical efficient preconditioners have—as mentioned in Section 3.1.3.3—only a small degree of data parallelism. Good preconditioners must respect the structure of the matrix (R, m).

From the point of view of the conjugate gradient method in Figure 7.9 a preconditioner is a class that has a method `solve(const Vector&, Vector&)`. In the next Section 7.2.3.3 it is shown how a simple parallel preconditioner can be defined in Janus.

7.2.3.3 A Simple Preconditioner

In this section it is shown how a simple preconditioner can be defined for the solution of the sparse system assembled in Section 7.2.2. The preconditioner in question implements *diagonal scaling* and uses the inverse diagonal entries of the matrix (R, m). This preconditioner is also referred to as *Jacobi preconditioning*[52].

As requested by the conjugate gradient method in Figure 7.9, there must be a method solve(const V&, V&) with an appropriate vector type V. In this example, the type Reals that was introduced in Section 7.2.1 is deployed.

Figure 7.10 shows the class Jacobi that implements the Jacobi preconditioner.

```
  class  Jacobi {
    Reals  inverse;
  public:
    Jacobi(const Relation& r, const Reals& m)
    {
      inverse.resize(r.size1());
      for(size_t i = 0; i < r.size1(); i++) {
        size_t first = r.first(i);
        for(size_t k = 0; k < r.size(i); k++)
          if (first == r.second(i,k)) { // diagonal entry
            Real diag = m[r.position(i,k)];
            inverse[i] = (diag == 0) ? 1.0 : 1.0 / diag;
            break;
          }
      }
    }

    void solve(const Reals& v, Reals& w) const
    {
      for(size_t i = 0; i < v.size(); i++)
        w[i] = v[i]*inverse[i];
    }
};
```

Figure 7.10: Implementation of Jacobi preconditioner

In the constructor of class Jacobi of Figure 7.10, the inverse of the diagonal of the matrix (R, m) is computed. At first, using the access methods from Table 4.15 it is determined at which positions of the coefficient vector the diagonal entries reside. Once a diagonal entry has been found, it is inverted (or set to 1 if the diagonal entry equals 0).

The method solve in Figure 7.9 just multiplies the input vector with the initially computed inverse diagonal entries.

7.2.4 Discussion

This section has covered many aspects of finite element programming. Firstly, it has been described how the basic sets and relations that constitute a finite element mesh can be represented with Janus components. It has been profoundly discussed how the initialization of these sets and the associated attributes is performed using the two-phase property of Janus data structures.

Secondly, the use of Janus when assembling element matrices to a large sparse matrix has been investigated. Interestingly, this computation follows a pattern that is similar to the initialization of sets and relation in Section 7.2.1. The use of two-phase algorithms allows to hide most aspects of an underlying distributed-memory architecture from the user.

Thirdly, it has been shown how an iterative solver—a key component of scientific software—is implemented in Janus. The `pull_matrix` method of Relation allows to utilize the underlying sparse matrix format efficiently. The implementation of other key components such as inner products or preconditioners has also been discussed. The performance of this Janus implementation is evaluated in Section 8.3.

7.3 Red-Green Triangle Refinement

In this section, the Janus implementation of a triangle refinement method is presented. Such a refinement scheme can be used, for example, when the finite element problem of Section 7.2 is to be solved on a sequence of adaptively refined triangulations.

To be more precise, the red-green refinement method whose subdivision rules were presented in Section 3.1.4.1 is discussed.

The discussion includes

- The outline of a `Triangulation` data structure (Section 7.3.1). This data structure will reuse most of the types introduced in Section 7.2.1. Additional relations to access data that are associated with the edges of triangles are introduced as well.
- A description of how the *closure* process of the initial edge marking

is implemented (see Section 7.3.3). Closure is—as discussed in Section 3.1.4.3—an iterative procedure that requires for each element the evaluation of data that are associated with its edges or faces. Here the nested `accessor` type that is provided by models of Relation is used to efficiently access these data regardless whether the mesh is distributed or not.

- The actual subdivision of the marked edges and triangles is outlined in Section 7.3.4. In order to avoid multiple insertions of triangles or edges into the new triangulation, an unambiguous subdivision policy for triangulations is formulated.
- Finally, it is explained in Section 7.3.5 how the subdivided triangulation is repartitioned to remove major imbalances that can occur during adaptive refinement. The point is: when using Janus components to represent a distributed mesh it is relatively easy to redistribute a mesh according to the mapping information provided by *external* repartitioning tools.

7.3.1 Representing the Triangulation

The `Triangulation` used in this section shall hold the data that are produced by the TRIANGLE mesh generator[93] These data include not only the sets of vertices or triangles but also associated properties. Thus restructuring the triangulation means not only that new triangulation elements, that is, triangles, edges, or vertices are generated and unnecessary ones are removed. It also means that the associated properties are properly defined, for example, interpolating coordinates of vertices that are created through subdivision of an edge.

Here is a complete list of the involved sets and associated properties as required by TRIANGLE.

1. The set of *vertices* of the triangulation where a vertex is represented by an integer. Associated attributes are:
   (a) the *Cartesian coordinates* of the vertices
   (b) for each vertex a flag that indicates whether it is a *boundary vertex* or not.
2. The set of *edges* of the triangulation where an edge is represented by an ordered pair of its vertices.

   (a) The only associated property of an edge is a flag that indicates whether it is a *boundary edge*.

3. The set of *triangles* of the triangulation.

The simple types `Vertex`, `Triangle`, `Flag`, and `Coordinate` from Section 7.2.1 together with their container equivalents `Vertices`, `Triangles`, `Coordinates`, and `Flags` are used to represent most of the above mentioned sets and associated properties.

7.3.1.1 Representing Edges

A new type `Edge` is introduced to represent edges of a triangulation.

```
typedef boost::array<Vertex,2> Edge;
```

Not surprisingly, the fixed-size container class `array` of Boost[28] is used to define `Edge`. However, the template class `std::pair` could have also been used.

Note that only *undirected* edges are considered. One way to deal with this requirement is to ensure that for each edge $e = (v_1, v_2)$ the condition $v_1 < v_2$ holds. Edges that are created using the function

```
Edge make_edge(const Vertex& v, const Vertex& w);
```

are guaranteed to fulfill this requirement.

As for the sets of vertices and triangle, the Janus domain class `sorted_domain` is deployed to represent sets of edges.

```
typedef  jns::sorted_domain<Edge>  Edges;
```

As in the case of vertices, boundary flags for edges will be represented by objects of type `Flag` and `Flags`.

7.3.1.2 Representing Red-Green Information

The rules of red-green triangle subdivision have been formulated in Section 3.1.4.1. As mentioned there, it is important to keep track of the *color* of triangles in order to ensure that triangles that have been created by halving an angle, that is *green triangles*, are not further subdivided.

Triangle colors can be represented by objects of type `Flag` using the symbolic constants

```
const Flag Red   = 0;
const Flag Green = 1;
const Flag Blue  = 2;
```

The color *Blue* serves as an additional color to differentiate the three edge of a green triangle.

It had been also pointed out in Section 3.1.4.1 that it is important to be able to reconstruct the red parent triangle of a two green sibling triangles. As it will be seen in the following subsections, this can be achieved by assigning to each edge one of the following flags:

```
const Flag First  = 0;
const Flag Second = 1;
const Flag None   = 2;
```

The meaning of these flags for an edge $e = (v_1, v_2)$ is

- `First`: it is a green edge and its *first* vertex v_1 subdivided the edge in Figure 3.7.
- `Second`: it is a green edge and its *second* vertex v_2 subdivided the edge in Figure 3.7.
- `None`: it is a red edge.

7.3.1.3 The Class `Triangulation`

The public access methods of the `Triangulation` data structure is shown in Figure 7.11.

Each of the public access methods of class `Triangulation` in Figure 7.11 returns a reference to a non-public data member. How these member can be initialized by reading data from a file has been explained in Section 7.2.1.3.

7.3.2 Additional Relations and Properties

Several additional data structures are necessary to be able to perform red-green refinement. However, as they are only temporarily needed, they are *not* members of class `Triangulation`.

```
class Triangulation {
public:
    const Vertices&    vertices() const;
    const Edges&       edges() const;
    const Triangles&   triangles() const;

    const Coordinates& coordinates() const;
    const Flags&       boundary_vertices() const;

    const Flags&       boundary_edges() const;
    const Flags&       edges_mode() const;
};
```

Figure 7.11: Public access method of class `Triangulation`

The necessary items include several *relations* and *properties* that are associated with triangles of the triangulation.

7.3.2.1 The Triangle-Edge Relation

The triangle-edge relation associates with each triangle the (global) positions of its three edges—see Figure 7.12. This relation plays an essential role in red-green refinement.

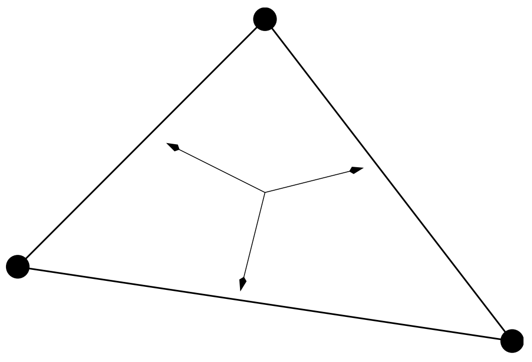

Figure 7.12: Triangle-edge relation

As the triangle-vertex `tv` relation of Figure 7.5 it will be represented by an object of type `Relation`. The corresponding relation object will be referred to as `te`. This relation object can be initialized analogously to the triangle-vertex relation in Figure 7.6. A function object that describes the relationship a triangle and its edges can be defined as:

```
struct  TriangleEdges
{
    typedef Triangle              argument_type;
    typedef boost::array<Edge,3>  result_type;

    result_type operator()(argument_type t) const
    {
        result_type edge;
        edge[0] = make_edge(t[1], t[2]);
        edge[1] = make_edge(t[0], t[2]);
        edge[2] = make_edge(t[0], t[1]);
        return edge;
    }
};
```

7.3.2.2 The Edge-Vertex Relation

Similar to the triangle-edge of Figure 7.12, the *edge-vertex* relation in Figure 7.13 is used during edge-green refinement. This relation is represented by the object `ev` of type `Relation`.

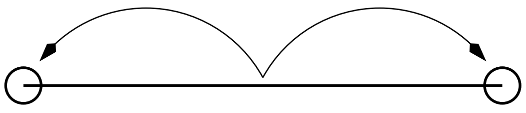

Figure 7.13: Edge-vertex relation

7.3.2.3 Triangle Colors and Edge Permutations

One immediate use of the triangle-edge relation is to compute for each triangle:

1. Its *color*, that is, whether it is *red* because it was created according to Figure 3.6 or whether it is *green* because it was created according to Figure 3.7.

2. In case of a green triangle, the three colors *red*, *green*, and *blue* can be assigned to its edges—see Figure 7.14. For the correct implementation of the red-green subdivision rules it is necessary to determine the permutation of these three colors with respect to the order in which the positions of the edges occur in the triangle-edge relation.

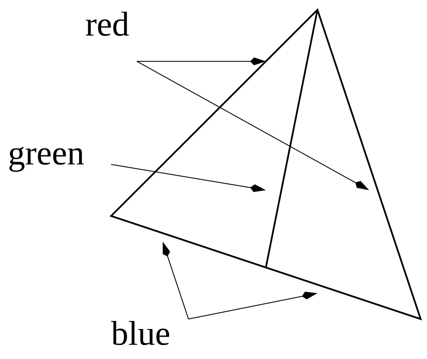

Figure 7.14: Edge colors of green Triangles

3. While traversing the triangle-edge relation and accessing the edges and associated data another important relation can be initialized. It is an edge-edge relation that associates with each green edge its corresponding two red edges as shown in Figure 7.15. This relation is useful when two sibling green triangles have to be removed because one of their edges has been marked for refinement.

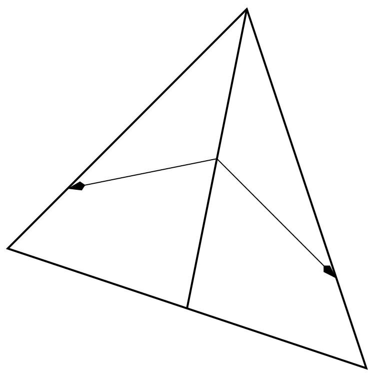

Figure 7.15: Edge-edge relation of green Triangles

All these information can be determined by *accessing* for each triangle its *edge mode* attributes. In order to compute the permutations the edges must also be accessed. These computations are performed by the routine `prepare_refinement` whose signature is shown here:

```
void prepare_refinement(const Edges&  edges,
                        const Flags&  edge_colors,
                     const Relation&  te,
                              Flags&  triangle_colors,
                   jns::size_vector&  red,
                   jns::size_vector&  green,
                   jns::size_vector&  blue,
                           Relation&  ee);
```

Note that the permutations for the edges of green triangles are represented by the three property functions `red`, `green`, and `blue` which are of type `jns::size_vector`.

The implementation of this routine is presented in Figure 7.16.

First the triangle-edge relation is bound to the edges and edge modes of the mesh. Then for each triangle the edge modes are evaluate using the function `check_flag`.

```
std::pair<size_t,Flag>
check_flag(size_t i, const Relation::accessor<Flag>& mode) {
  for(size_t k = 0; k < 3; ++k)  {
    Flag f = mode.get(i,k);
    if (f == First || f == Second) return std::make_pair(k,f);
    else assert(f == None);
  }
  return std::make_pair(3,None);
}
```

This function returns a pair of `size_t` and `Flag`. The first component indicates which of the edges is a green edge. The second component holds the flag of that edge. If no green edge exist for this triangle then the first component is set to 3.

In case the *i*-th triangle in Figure 7.16 has a green edge it is checked which of the remaining edges coincides with the vertex indicated by the flag of the green edge.

```
Relation::accessor<Flag> em_access(te, edge_mode);
em_access.pull();
Relation::accessor<Edge> edge_access(te, edges);
edge_access.pull();

for(size_t i = 0; i < te.size1(); ++i)
{
   assert(te.size(i) == 3);
   std::pair<size_t,Flag> which = check_flag(i, em_access);
   if (which.first == 3)  triangle_colors[i] = Red;
   else {
      triangle_colors[i] = Green;
      green[i] = which.first;
      const Edge edge = edge_access.get(i,green[i]);
      const Vertex v = edge[which.second];
      for(size_t k = 0; k < 3; ++k)
         if (k != green[i]) {
            Edge e = edge_access.get(i,k);
            if(e[0] == v || e[1] == v)  blue[i] = k;
            else                        red[i]  = k;
         }
      ee.insert(te.second(i,green[i]), te.second(i,red[i]));
   }
}
ee.freeze();
```

Figure 7.16: Accessing edges and edge modes from triangles

7.3.3 Closure of Edge Refinement Marks

The process of closing a refinement is as pointed out in Section 3.1.4.3 an iterative procedure. The Janus implementation for the case of red-green subdivision rules is shown in Figure 7.17.

The function `closure` modifies its last argument `marks` which is a property function on the edges of the mesh that represents the refinement marks as boolean values. A value of `true` indicates that the edge is to be refined.

```
void closure(const Edges&   edges,
             const Relation&   te,
                 const Flags&   triangle_colors,
      const jns::size_vector&   red,
      const jns::size_vector&   green,
      const jns::size_vector&   blue,
                       Flags&   marks);
```

The remaining arguments are the edge set `edges`, the triangle-edges relation `te`, the triangle colors (either red or green), and the permutations of edge colors of green triangles.

In order to evaluate the edge refinement marks in each triangle the triangle-edge relation `te` is used to declare the edge mark accessor object `em`. At the beginning of each iteration, the accessor object is updated by calling its `pull` method.

During each iteration, every triangle *evaluates* its edge refinement marks using the accessor `em` and *modifies* them for according to the rules of red-green refinement.

A position accumulator (§5.5.2.2) object `accu` is deployed in order to modify the refinement marks on the edges of a triangle. The concrete type of this accumulator is:

```
jns::position_accumulator<Edges,Flag,std::logical_or<Flag> >
```

This indicates that `Flag` values inserted at the same position of the property function `edge_marks` to which the accumulator is bound are reduced by the *logical or*, that is, the operator `||`.

The positions that are put into the accumulator are delivered by the triangle-edge relation `te`. The inline function `update` is used to hide the details of calling the accumulators `insert` method the rest of code in Figure 7.17.

```
void
update(const Relation& te, size_t i, size_t k, accu_t& accu)
{
    accu.insert(te.second(i,k), true);
}
```

Note that calling `update`, that is the `insert` method of `position_accumulator` only indicates the modification. The modification has been definitely performed when the `freeze` method has been called.

```
Relation::accessor<Flag> em(te, marks);
for(size_t cnt = 0; cnt < jns::size(edges); )
{
  em.pull();

  accu_t accu(edges, marks.begin(), marks.end());
  for(size_t i = 0; i < te.size1(); ++i) {
    if (triangle_colors[i] == Red) {
       Flag sum = 0;
       for(size_t k = 0; k < te.size(i); ++k)
          if (em.get(i,k)) sum++;
       if (sum > 1) for(size_t k = 0; k < te.size(i); ++k)
          update(te, i, k, accu);
    }
    else {
       assert(triangle_colors[i] == Green);
       if (em.get(i,blue[i]) || em.get(i,green[i]))
          update(te, i, red[i], accu);
       if (em.get(i,blue[i]) || em.get(i,red[i]))
          update(te, i, green[i], accu);
    }
  }
  accu.freeze();
  size_t cnt1 = jns::accumulate(marks.begin(),marks.end(),0);
  if(cnt1 == cnt) break;
  cnt = cnt1;
}
```

Figure 7.17: Closure of refinement marks

For red triangles, the refinement rules in Section 3.1.4.1 state the a modification of the edge marks of a *red* triangle only occurs if more than one edge has been marked. Thus the code for red triangles in Figure 7.17 first counts the number of marked edges and marks all of them in case more than one has been marked.

For green triangles the rules are more complex and require to evaluate the edge marks dependent on their color with respect to triangle—see Figure 7.17. Figure 7.18 shows the modification rules. A • indicates a refinement mark whereas a ∘ indicates edges to be refined.

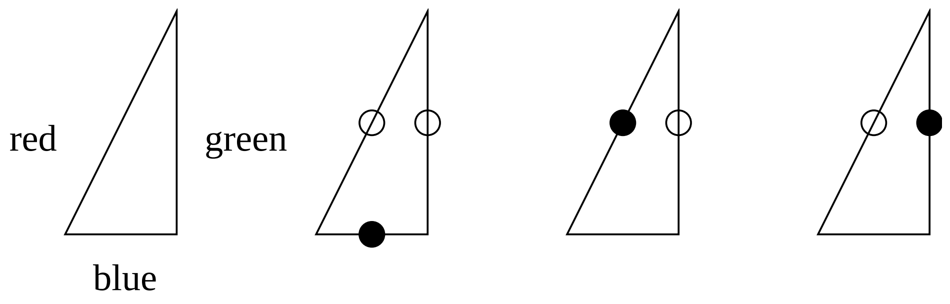

Figure 7.18: Modification rules for edge refinement marks of green triangles

A mark on a blue edge triggers the green and red edges for refinement. Green and red edges, on the other hand do not concern the blue edge. They only trigger each other.

After calling the `freeze` method of the accumulator, the indicated modifications of the edge refinement marks are properly updated.

The function `jns::accumulate` counts the number of marked edges. This number increases with every iteration (see Section 3.1.4.3). If the this number does not change from one to another iteration than the closure process is completed and the iteration loop can be terminated.

7.3.4 Subdivision of Marked Edges and Triangles

Once the refinement marks have been closed, the edges and triangles can be subdivided. This mesh refinement is performed by creating a new mesh that contains the unmarked vertices, edges, and triangles from the old triangulation and in addition the new mesh components that are created by subdivision of old components. However, not only the new sets of vertices, edges, and triangles must be correctly created. The associated attributes, that is the vertex coordinates, boundary and edge mode flags of the new triangulation must be correctly defined. The latter process includes interpolating the coordinates of newly created vertices.

Two instantiations of `collector` are use in order to temporarily hold the attributes associated with the newly created vertices and edges.

```
typedef jns::collector<Vertex,VertexData>  vertex_collector;
typedef jns::collector<Edge,EdgeData>      edge_collector;
```

The auxiliary type `VertexData` is defined as pair of `Coordinate` and `Flag`. The type `EdgeData` is defined as `std::pair<Flag,Flag>` which reflects the fact there are two `Flag`-valued property functions (`boundary_edges` and `edges_mode`) associated with the edges.

When evaluating the refinement marks and applying the corresponding subdivision rules, care must be taken that newly created mesh components are inserted in their data structures only once. This restriction is a basic requirement of the concept Two-Phase Domain of Section 4.2.2. A basic mean to achieve this is to deal with vertices, edges, and triangles separately.

7.3.4.1 Subdivision of Vertices

Strictly speaking, vertices are not subdivided. Rather all vertices of the old triangulation are put into the new one. The routine `subdivide_vertices` in Figure 7.19 *copies* the vertices and the associated attributes (coordinates and boundary flags) into a *domain collector*. Thus, mesh refinement uses essentially the same data structures as for initialization of a a mesh—see Section 7.2.1.3.

```
void  subdivide_vertices(const Triangulation&  mesh,
                         vertex_collector&     vcoll)
{
  for(size_t i = 0; i < mesh.vertices().size(); ++i)
  {
    Vertex      v = mesh.vertices()[i];
    VertexData  a(mesh.coordinates()[i],
                  mesh.boundary_vertex()[i]);
    vcoll.insert(v, a);
  }
}
```

Figure 7.19: Copying old vertices and their attributes

Note that the collector is not frozen. Thus, vertices that are created by subdividing edges can be inserted later.

7.3.4.2 Subdivision of Edges

Subdivision of edges is more complex because their *mode* (Section 7.3.1) must be taken into account. Three cases can occur which are listed below. The Janus implementation is shown in Figure 7.20. This function has the signature.

```
void subdivide_edges(const Triangulation&  mesh,
                             const Flags&  edge_marks,
                            const Vertex&  vmax,
                          const Relation&  ev,
                          const Relation&  ee,
                        vertex_collector&  vcoll,
                          edge_collector&  ecoll,
                               Triangles&  new_triangles);
```

Note that in addition to the vertex collector `vcoll`, there is also an edge collector `ecoll` with `EdgeData` as second template parameter. Since there are no attributes associated with the triangles there is no need to use the `collector` template. Newly created triangles or triangles that are copied from the current triangulation can directly be inserted into the `Triangles new_triangles`.

The meaning of the other arguments are explained in conjunction with the different subdivision cases.

1. A *marked* edge $e = (u, v)$ with the attribute `None`, that is an edge that is not shared by two sibling triangles is subdivided into a new vertex w that represents the midpoint of the edge and the two edges (u, w) and (v, w)—see Figure 7.21.

   A unique name, that is a unique integer value, must be assigned to the newly created vertex w. This name must be different from all vertices of the current triangulation and different from any other vertex name created during this subdivision. One way to achieve this is the following rule to name w

   $$\text{maximum of old vertices } + 1 + \text{global position of edge } (u, v). \quad (7.3)$$

   The following piece of code uses the parallel STL algorithm `jns::accumulate` (§5.5.3) to compute the maximum of the vertex set `vertices`.

```
Relation::accessor<Coordinate>  coord(ev, mesh.coordinates());
for(size_t i = 0; i < mesh.edges().size(); ++i) {
   const Edge e = mesh.edges()[i];
   const Flag b = mesh.boundary_edge()[i];
   const Flag emode = mesh.edge_mode()[i];
   if(edge_marks[i] == true)  {
     if (emode == None) {
       Vertex vn = new_vertex(mesh.edges(), vmax, i);
       Coordinate vcn = 0.5*(coord.get(i,0) + coord.get(i,1));
       vcoll.insert(vn, make_pair(vcn,b));
       ecoll.insert(make_edge(e[0],vn), make_pair(b,None));
       ecoll.insert(make_edge(e[1],vn), make_pair(b,None));
     }
     else {
       assert((emode == First) || (emode == Second));
       Vertex x  = new_vertex(vmax, ee.second(i,0));
       Vertex y  = new_vertex(vmax, ee.second(i,1));
       Vertex u  = e[emode];
       Vertex v  = e[1-emode];
       new_triangles.insert(make_triangle(u, x, y));
       new_triangles.insert(make_triangle(v, x, y));
       ecoll.insert(make_edge(u, x), make_pair(false,None));
       ecoll.insert(make_edge(u, y), make_pair(false,None));
       ecoll.insert(make_edge(x, y), make_pair(false,None));
     }
   } else ecoll.insert(e,make_pair(b,emode));
}
```

Figure 7.20: Subdivision of Edges

```
Vertex vmax = jns::accumulate(vertices().begin(),
     vertices().end(), 0, jns::maximum<Vertex>());
```

The value 0 serves here as the initial (neutral) element for the reduction with the binary function object `jns::maximum<Vertex>`.

In order to insert the new vertex w into the vertex collector of Figure 7.19, its coordinates and its boundary flag must be determined. The latter is quite simple because the newly created vertex is a boundary vertex if and only if the edge (u, v) was a boundary edge. The

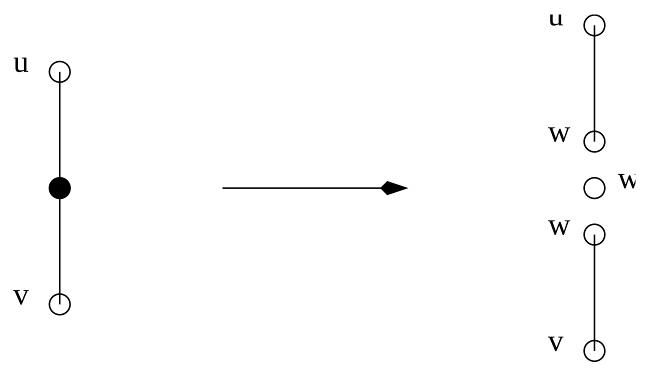

Figure 7.21: Simple subdivision of an edge

coordinates of w are computed are interpolated by forming the arithmetic mean of the coordinates of u and v. In order to access the coordinates of u and v from the edge (u, v) the edge-vertex relation `ev` of Figure 7.13 is used.

The newly created edges (u, w) and (v, w) have the same boundary flag and mode flag as their parent edge (u, v). Both edges are inserted into the domain collector for edges introduced above.

2. An marked edge with edge mode `First` or `Second` is shared by two sibling triangles and *not* subdivided. Rather the *red* parent triangle is regularly refined as shown in Figure 3.6 Thus this edge triggers the creation of new triangles.

   In order to avoid multiple inserts of edges or triangles, this kind of edge inserts only those edges and triangles in Figure 7.22 that are drawn with solid lines. In other words the two red triangles (u, v, x) and (u, v, y) and three the edges (v, x), (v, x), (u, x), and (u, y) are inserted. These other triangles and edges are inserted while subdividing green triangles and their edges.

   The newly created edges are inserted into the edge collector. Obviously, these edges can't be boundary edges and their edge mode is `None`.

   The names of the vertices x and y are determined according to formula 7.3. The edge (u, v) accesses the global positions of the edges on which x and y were created by using the edge-edge relation of Figure 7.15

3. Unmarked edges are copied together with there associated attributes into the edge collector `ecoll`.

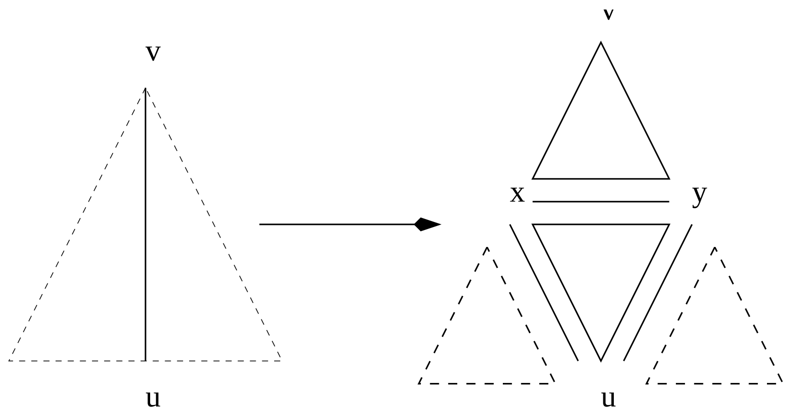

Figure 7.22: Edges and triangles created by marked green edges

7.3.4.3 Subdivision of Triangles

Five cases must be considered when subdividing triangles. The closure process described in Section 7.3.3 ensures that more cases do not occur. Similar to the subdivision of edges, it is exactly described which objects are created during subdivision. This is necessary to ensure that no object is inserted more than once into a two-phase domain.

The emphasis of this subsection is on the explicit description of responsibilities when creating new triangulation elements or copying old ones.

1. Unmarked (red or green) triangles are copied into the new triangle set.
2. A red triangle with exactly one marked edge is subdivided into two green triangles and the edge that is shared by them—see Figure 7.23. The triangles are inserted into the new triangle set and the edge into the edge collector.

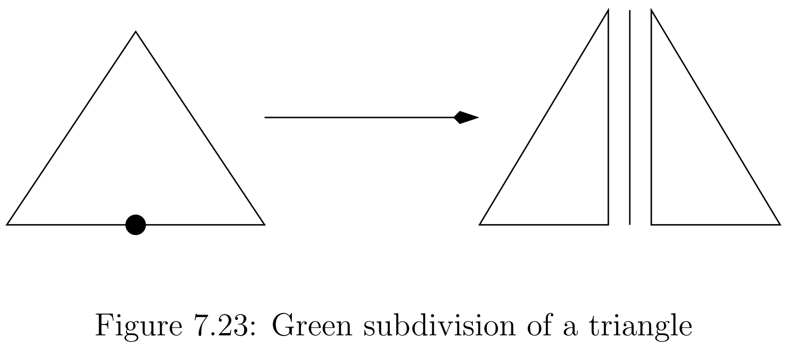

Figure 7.23: Green subdivision of a triangle

3. A red triangle with three marked edges is subdivided into four triangles and three edges as shown in Figure 7.24.

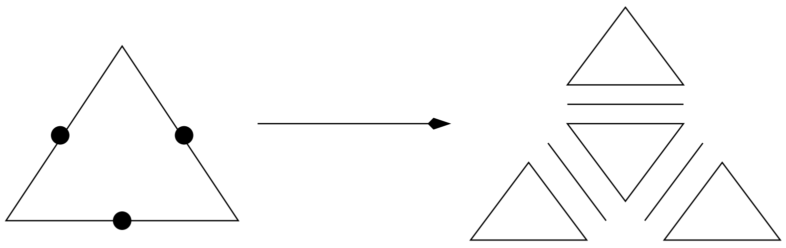
Figure 7.24: Red subdivision of a triangle

4. If a green triangle has two marked edges then the small red triangle that is indicated by the dotted line in the left part of Figure 7.25 is inserted in the new triangle set.

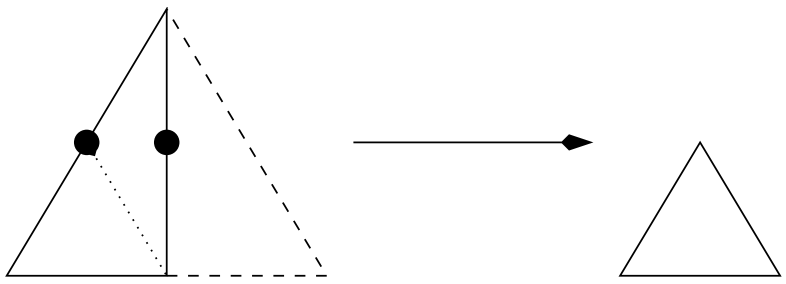
Figure 7.25: Subdivision of a green triangle with two marked edges

5. If a green triangle has two marked edges then the small red triangle that is indicated by the dotted line is subdivided into two green triangles and the edge shared by them—see Figure 7.26.

7.3.4.4 Completing the Subdivision

After the three subdivision routines for vertices, edges, and triangles have been called the collectors for vertices, edges, and the new triangle set contain all data to initialize an new `Triangulation` object. The collectors `vcoll` and `ecoll` are frozen and then traversed as shown in Figure 7.4 to initialize the new vertices and edges and there associated attributes. The new triangle set need only to be frozen since there are no attributes associated with it.

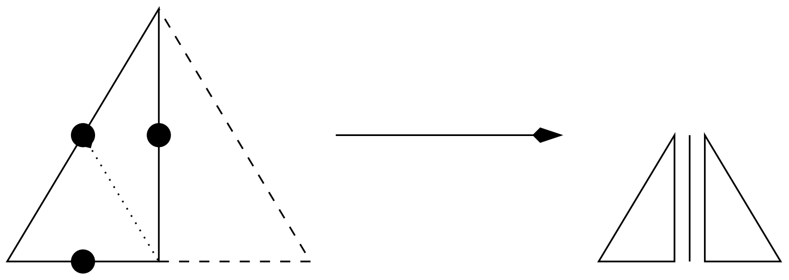

Figure 7.26: Subdivision of a green triangle with three marked edges

7.3.5 Repartitioning of Refined Meshes

The vertices, edges, and triangles that were put into the collectors and domains using the corresponding `insert` method. This implies that, for example, the newly created triangles and edge of Figure 7.23 are placed at the same process where their parent triangle was located. After several mesh restructuring step this can lead to severe load imbalances that affect the performance of the parallel program.

For this reason it important to check the distribution of the domains regularly and perform a repartitioning of the mesh if it is necessary. After a new partitioning has been determined, the vertices, edges, and triangles together with their associated attributed are inserted in their collectors. However, this time the method `insert_at` that allows the specification of mapping information is used.

The load of a domain is return by the global template function `jns::load` of Section 5.6.2.3. In order to maintain a good parallel performance a load of less than 1.1 is usually sufficient.

7.3.5.1 Combining Remapping with Subdivision

It is also possible to perform subdivision and remapping simultaneously. That is, instead of creating a new triangulation and then remapping it, one can use the refinement marks to determine *weights* on the old triangulation, compute a new mapping for the weighted triangulation. and insert the new vertices, edges, and triangles into the collector using this mapping information.

This approach has a performance advantage *if* remapping occurs very often. A disadvantage of this approach, from the point of view of software-engineering, is the tight coupling of the details of the mesh-restructuring algorithm with the concern to balance the computational load.

7.3.5.2 Graph Partitioners and Geometric Partitioners

Many partitioning algorithms evaluate the graphs, which are inherent to the application structure. In mesh-based applications mesh entities such as the vertices or triangles of a Triangulation can be considered as vertices of a graph. A dependence between two entities can be expressed as an edge of the graph connecting the associated vertices. The set of vertices and their nearest-neighbor relation `r` of the triangulation in Section 7.2.2 is an example of such a graph.

Graph partitioning tools applied to these graphs provide load balancing and reduce the amount of data to be communicated, e.g. by minimizing the *edge cut*. METIS[63] is an example of a graph partitioner.

It is, however, not mandatory to use graph information explicitly in order to achieve a good partitioning. In most mesh-based applications mesh points are labeled with their spatial coordinates, and dependences exist only between geometrical neighboring mesh points. For these meshes, load balancing and implicit edge cut minimization can be achieved by a suitable geometric clustering of the mesh points. MOSAIK[103] presents an example of a geometric mesh partitioner. MOSAIK is particular well-suited for repartitioning of refined meshes. It does not need to start from scratch like other algorithms and completely reuses the partitioning of the previous mesh.

7.3.6 Discussion

Adaptive mesh-refinement has been a key motivation for the development of the Janus framework. For this reason, the implementation of red-green triangle refinement has been discussed so carefully. The complexity of the description is in the problem itself and not an artifact of the Janus implementation.

A key component of mesh refinement is the correct evaluation of the refinement marks in each element. Here, the nested `accessor` type that is provided by `Relation` has been used. Accessors are used together with `accumulator` objects to evaluate and update the refinement marks.

Load balance is an important issue on a parallel platform. In order to ensure a balanced load the newly created mesh must usually be repartitioned from time to time. The task of computing a balanced partition is beyond the scope of Janus. However, using the `insert_at` method of Distributed Domain Janus can easily incorporate mapping information that was computed during the runtime of a parallel mesh refinement algorithm.

7.4 The Bellman-Ford Algorithms

In this section a generic Janus implementation of the Bellman-Ford algorithms is discussed. The basic idea of the Bellman-Ford single-source shortest path algorithm have already been described in §3.1.5.1. Its key component is the parallel *relaxation* over all edges expressed by Equation 3.16.

This means that the weights on the edge and the distance on the vertices are combined in a way that is similar to a matrix-vector multiplication. However, an important difference is that the result of a relaxation is not only a combination of the edge weights and vertex distances. Rather for each vertex the nearest vertex that is its neighbor must be selected. For this reason, the `pull_matrix` matrix method is not the best choice and similar to the case of adaptive mesh refinement in Section 7.3 the nested `accessor` type of Relation is used to access and evaluate individual values.

Figure 7.27 shows the source of the template function `bellman_ford`.

7.4.1 Interface

Template Parameters The three template parameters of `bellman_ford` are

`Edges` which must be a model of Relation,

`Nodes` which must be a model of Domain, and

`Type` which typically is a built-in integral or floating point type. However also a user-defined type can be used if it provides addition through `operator +` and comparison by `operator <`.

Return and Argument Values The function `bellman_ford` returns the Boolean value `true` if the *tree* that represents the shortest paths from a single source node to all nodes has been constructed. In case a *negative cycle* (due to negative weights) has been found, the Boolean value `false` is returned.

```
template<typename Edges, typename Nodes, typename Type>
std::pair<bool,size_t> bellman_ford(const Edges& e,
     const Nodes& n, const Type* w, std::vector<Type>& d,
     size_t* p)
{
  typedef volatile Type vT;
  typename Edges::accessor<Type> access(e,d.begin(),d.end());
  for(size_t it = 0; it < jns::size(n); ++it)
  {
    jns::accumulator<bool,std::logical_and<bool> > ok(true);
    access.pull();
    for(size_t i = 0; i < n.size(); ++i)
    {
      size_t min_k = e.size(i);
      vT relaxed = d[i];
      for(size_t k = 0; k < e.size(i); ++k) {
        vT t = w[e.position(i,k)] + access.get(i,k);
        if (t < relaxed) {
          relaxed = t;
          min_k = k;
        }
      }
      if (min_k < e.size(i)) { // something happened
        d[i] = relaxed;
        p[i] = e.second(i, min_k);
        ok.insert(false);
      }
    }
    if (ok.freeze()) return std::make_pair(true,it);
  }
  return std::make_pair(false,jns::size(n));
}
```

Figure 7.27: Janus implementation of Bellman Ford algorithm

The arguments `e` and `n` are the relation and the node set of the graph, respectively. The third argument `w` points to an array of `e.size()` objects of type `Type` that represent the *weights* on the graph edges.

The arguments `d` and `p` point to arrays of `n.size()` objects of type `Type` and `size_t`, respectively. On successful completion of `bellman_ford`, the array p holds for each node the (global) position of the node's predecessor in the tree constructed by the Bellman-Ford algorithm. The array d, on the other hand, will hold the distance to the single source.

Note that the single source $s \in n$ must be properly marked in the array d by assigning *appropriate high* distances all nodes but at s (see §3.1.5.1). As initial value *nil* for the elements of array p and to identify the root of the tree serves the number `jns::size(n)` (§5.6.2.1) since this value cannot be a (global) position[2] of a node in n. Thus, the arrays d and p are both input and output parameters.

7.4.2 Implementation

The typedef in Line 5 of Figure 7.27 is a shortcut to add the *volatile* specifier to `Type`. It is used to prevent aggressive[3] optimizations of the comparison in Line 14 that may lead to incorrect results[99, p. 808].

Setup In Line 6 the accessor for the property function d is declared.

The Bellman-Ford algorithm needs to perform at most as many iterations as there are nodes in the graph. The call of the global function `jns::size` (Line 7) of Section 5.6.2.1 returns this information.

The algorithm can be terminated when no relaxations (Equation 3.16) need to be performed anymore. In order to keep track of changes in one iteration, the Janus `accumulator` template can be used (see §5.5.2.1).

Relaxation Sweep The Lines 11 to 27 cover one sweep of relaxations for all nodes of the graph. The sweep is performed by relaxing all edges that belong to a certain node (Line 11). Note that instead of `n.size()` one could also call `e.size1()`.

However, in order to to be able to access values of d through the accessor `access` they must be cached (Line 10) using the accessor's `pull` methods (Table 4.18).

[2]Note that for a domain object `dom`, the global function `jns::size(dom)` returns the number of *all* elements. Only for a non-distributed domain `jns::size` returns the same as result the method `domain.size()`. Thus `jns::size` provides unified access to an important graph property, regardless whether it is distributed or not.

[3]In particular floating point types are affected by this problem since values kept in register sometimes use more bits than those kept in variables. For example, Intel's IA-32 architecture provides 80-bit register to store 64-bit numbers.

In Lines 13 through 21, the (relative) edge number `min_k` that yields the largest relaxation gain for the i-th is determined. If such an edge has been found (Lines 22 through 26) then the following operations are executed.

Line 23 The relaxation along the edge `min_k` is performed and `d` is updated.

Line 24 The predecessor of the i-node is set to the (global) position of the node to which the edge refers.

Line 25 It is marked that a relaxation has been performed.

Note that the new value of `d[i]` is only guaranteed to be accessible through `access` if `pull` has been called. The Bellman-Ford algorithm however does not depend on an immediate accessibility of the newly computed value `d[i]`. Thus, during each relaxation sweep `pull` must be called only once which underlines the *robustness* of Bellman-Ford. On the other hand, this makes the number of relaxation sweeps dependent on the distribution of the graph.

Termination The `freeze` operation in Line 28 returns the reduction of all update marks. If no update did occur during the current sweep that the algorithm terminates successfully. The return statement in Line 30 indicates that a negative cycle has been encountered.

7.4.3 Discussion

The Bellman-Ford shortest path algorithm is an example of a data parallel graph algorithm that can be easily implemented with Janus. The `pull` and `get` methods of the nested `accessor` type of the Relation concept are sufficient to hide the most important source of communication in this algorithm. The two-phase property of the `accumulator` also hides communication that is related with the reduction operation.

Thus, the Janus concepts and components are expressive enough to implement parallel graph algorithm. The performance of this Bellman-Ford implementation is evaluated in Section 8.4. On the other hand, the implementation of Bellman-Ford that is provided by the Boost Graph Library[67] does not allow a straightforward parallel implementation on a distributed memory platform.

7.5 Conclusions

The Janus conceptual framework and its components provide a small yet expressive set of (distributed) data structures, algorithms, and parallel data access primitives. Most of these primitives rest on basic components of parallel algorithms like matrix-vector multiplication and can be found in other libraries as well. Janus provides these primitives in a generic way so that they can be easily customized by user-defined components.

The examples in this chapter have demonstrated that parallel programming relatively easy within the Janus framework. In particular, most of the data movement operation that result into communication on a distributed-memory architecture are hidden within higher level constructs, such as `pull` or `freeze`. However, a user must of course ensure that the requirements imposed by Janus concepts and components are properly fulfilled.

For example, it is not allowed to insert an element into a domain more than once (see Table 4.5). Therefore a Janus application must organize its insertion operation in a way that multiple insertions do not occur. Also a Janus application must explicitely declare the relation it will use to access (remote) data associated with a domain.

For these reasons the following three questions have been formulated. Answering these question during the design of a parallel Janus application greatly simplifies the correct and efficient use of Janus components.

1. Which object is responsible to manipulate, and in particular, to insert a certain object.
2. Which data are necessary to manipulate an object?
3. How can the necessary data be obtained?

In the following chapter, the performance of selected Janus applications is evaluated for various computing platforms.

Chapter 8

Performance Measurements

The previous chapters introduced the Janus architecture and its main components. In Chapter 7 the Janus architecture has been applied for the implementation of representative sample problems in the domain of data parallel applications. Thereby it has been proved that Janus is expressive enough to support a wide range of scientific application—ranging from cellular automata on rectangular grids to adaptive finite element methods and parallel graph algorithms.

In this chapter the performance of Janus applications is evaluated. In particular, it is investigated how Janus programs behave on a state-of-the-art parallel platform, namely a Linux cluster connected by dedicated high-performance network. Here a configuration of Janus is used that utilizes MPI for the implementation of its distributed components—see Chapter 6. The test platform is described in more detail in Section 8.1.

Performance evaluation has many aspects, including

- Comparing Janus on different platforms,
- Comparing Janus programs with implementations that use other frameworks,
- Examining the scalability of Janus on parallel platforms.

Not all aspects can be thoroughly investigated within the scope of this dissertation. However, an attempt is made to be as profound as possible for three of the applications described in Chapter 7. The three applications are *Conway's Game of Life* §7.1, *linear triangle element analysis* §7.2, and the *Bellman-Ford Shortest Path Algorithm* §7.4. Due to the lack of access to a reliable shared-memory system only the performance on sequential and distributed-memory systems can be compared. The performance of these codes is also compared with that of analogous implementations using state-of-the-art libraries in the respective domains.

8.1 Description of Test Platform

For performance evaluation a Linux PC cluster with 32 processors is used. Its specifications read:

- 16 Tyan Tiger MP S2460 boards where each board is equipped with two Athlon 1.2 GHz MP,
- 1GB RAM PC2100 and 20GB Local Disk per board,
- Myrinet 2000 network and 100 MBit FastEthernet interconnection network,
- Linux 2.4.17-smpSCORE operating system.

Myrinet[17] is a high-performance, packet-communication and switching technology that provides high data-rate and low-latency communication between host processes. The Myrinet network and the shared memory on each board is utilized by the SCORE[59] cluster operating system developed by Real World Computing Partnership. The SCORE system includes an implementation of MPI that is used by the generic MPI layer of Janus/Jade.

The Gnu C++ compiler g++[18] (the unofficial release 2.96 that is included in the Red Hat Linux distribution) will be used for all examples. The compiler flags `-O3`, `-funroll-loops`, and `-fexpensive-optimizations` are used as optimization flags. These flags perform among others the following optimizations: function inlining, loop unrolling for loops whose number of iterations can be determined at compile time or run time, and register renaming.

All times are measured with the utility function `jns::time()`. In a sequential configuration, this function is implemented using the C-library function `gettimeofday`. Parallel configuration on top of MPI use the function `MPI_Wtime` to implement `jns::time()`.

8.2 Game of Life

An implementation of Conway's Game of Life that uses the `grid` and `stencil` classes of Janus has been presented in Section 7.1. In this section, two aspects of the performance of this program are investigated.

1. The runtime of a (sequential) C implementation of *Life* is compared with the Janus program that is configured for a sequential machine.

2. It is investigated how the same Janus program, however, this time configured for a distributed-memory architecture, *scales* when the number of used processing elements increases.

The sequential C implementation of *Life* uses dynamically allocated integer arrays to represent the state of the cells. The code is shown below. The argument `matrix_size` denotes the length (measured in cells) of a square grid on which the computation is performed. The variable `ntimes` is the number of generations that are simulated.

```
void life(int matrix_size, int ntimes)
{
  int     i, j, k, sum ;
  int     **a, **b, **tmp ;
  /* initialization not shown */

  for (k = 0; k < ntimes; k++) {
    for (i = 1; i < matrix_size-1; i++) {
      for (j = 1; j < matrix_size-1; j++) {
        sum = a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] +
              a[i][j-1]   +             a[i][j+1]   +
              a[i+1][j-1] + a[i+1][j] + a[i+1][j+1];
        if ((sum == 2) && a[i][j]) b[i][j] = 1;
        else if (sum == 3)         b[i][j] = 1;
        else                       b[i][j] = 0;
      }
    }
    tmp = a; /* Swap data */
    a = b;
    b = tmp;
  }
}
```

The following Table 8.1 shows the time in seconds needed by both the Janus and the C program for 1000 generations on different grid sizes. Note that the measured times do not include the initialization of the arrays that hold the state of the cells. All times are measured in seconds.

	1000×1000	2000×2000	3000×3000
Time C Program	27.4	108.5	243.9
Time Janus Program	29.0	115.7	256.3
Overhead Ratio	**1.06**	**1.07**	**1.05**

Table 8.1: Relative overhead of a sequential Janus implementation of *Life* compared with a C implementation

The data of Table 8.1 show that the ratio of the runtimes of the Janus to the C implementation is approximately 1.06. This is a very good factor and shows that a modern C++ compiler can generate very efficient object code from template classes and functions.

The following two Tables 8.2 and 8.3 show timing results and the relative speedup for different usage scenarios of the SCORE cluster. Here, of course, the `grid` and `stencil` classes are configured to utilize MPI. The program is run on a medium-size grid of 2000×2000 points. Again, 1000 iterations are computed.

Table 8.2 shows the results when using only one of the two processors per board. This means that only 16 of the 32 processors can be used. Except for the step from 1 to 2 processors the runtime is reduced by roughly a factor of two when doubling the number of processors. The relative low speedup of 1.3 when using two instead of one processor is caused by extra copy operations in the implementation of the `pull` method of `stencil`. Future revisions of `stencil` have to remove this bottleneck.

Processors	1	2	4	8	16
Time	114.7	86.1	43.0	22.0	11.5
Relative Speedup	1.0	1.3	2.6	5.2	10.0

Table 8.2: Runtime results for parallel Janus *Life* program using *one* processor per board

Table 8.3 shows the results when using two processors per board (where possible). This has, on the hand, the advantage that all 32 processors of the

Processors	1	2	4	8	16	32
Time	114.7	97.5	54.5	27.4	14.5	7.9
Relative Speedup	1.0	1.2	2.1	4.1	7.9	14.5

Table 8.3: Runtime results for parallel Janus *Life* program using *multiple* processors per board

cluster can be utilized. On the other hand, the speedup results are not as good as in Table 8.2. For example, for 16 processors the relative speedup when using only one processor per board is 10, whereas in the case of two processors per board the speedup is approximately 8.

8.3 Finite Element Program

In this section the performance of a simple finite element solver implemented in Janus is evaluated. More precisely, the process of solving a the linear system of equations that results from a simple finite element approximation by a preconditioned iterative method is investigated.

In order to evaluate the performance of the Janus implementation of the conjugate gradient method in Figure 7.9 (p. 135) it is compared with the conjugate gradient routine of the PETSc package discussed in Section 2.4.3.1. In both cases, diagonal scaling is used as preconditioner.

The finite element problem under consideration is constituted by the Poisson problem discretized with linear finite elements. The details of this problem have been presented in the Sections 3.1.3 and 7.2.

The triangulation on which the Poisson equation was discretized consists of

- 345,909 vertices,
- 1,034,618 edges, and
- 688,709 triangles.

For linear finite elements the number of vertices is approximately the number of unknowns of the linear system. The number of edges indicates the number of non-zero entries in the sparse matrix. Thus, the matrix has approximately 10^6 non-zero entries.

The time for 500 iterations of the conjugate gradient method is measured for both the Janus implementation and the PETSc routine for various numbers of processors.

Table 8.4 shows the measured time in seconds and the relative speedup when using only one processor per board. Up to 8 processors, the Janus routine is slightly faster than the PETSc routine. In both cases, an almost ideal (relative) speedup is achieved and even surpassed in the case of PETSc. For 16 processors, Janus achieves a speedup of 15.6 and PETSc a speedup of 17.9.

Processors	1	2	4	8	16
Time Janus	77.9	40.8	20.1	9.9	5.0
Relative Speedup	1.0	1.9	3.9	7.9	15.6
Time PETSc	80.4	41.2	20.3	10.1	4.5
Relative Speedup	1.0	2.0	4.0	8.0	17.9

Table 8.4: Time and speedup of different conjugate gradient implementations when using one processor per board

Table 8.5 shows the measured time and the relative speedup when using two processors per board. As in the previous section, step from using one to two processors yields a worse speedup. Later, however, both routines scale remarkably well. For 32 processors, Janus achieves a speedup of 20.5 and PETSc a speedup of 28.7.

Processors	1	2	4	8	16	32
Time Janus	77.9	51.3	24.5	13.1	6.7	3.8
Relative Speedup	1.0	1.5	3.2	5.9	11.6	20.5
Time PETSc	80.4	53.3	26.6	14.4	6.5	2.8
Relative Speedup	1.0	1.5	3.0	5.6	12.4	28.7

Table 8.5: Time and speedup of different conjugate gradient implementations when using two processors per board

It is not easy to explain the differences in the scalability of the Janus and PETSc implementations. Sparse matrix-vector multiplication and inner products are the causes of communication in this preconditioned conjugate gradient method. Janus solely uses *collective* MPI functions whereas PETSc also utilizes collective operations such as `MPI_Allreduce`. However, many data exchange operations are implement by non-blocking send and receive operations (`MPI_Isend` and `MPI_Irecv`) in conjunction with an explicit synchronization (`MPI_Waitall`). This allows a fine tuned overlapping of communication and computation and may be one reason for the better scalability.

One also has to take into account the much longer availability and maturity of the PETSc software. It has been actively developed by a large team for at least 6 years. More than one hundred sample applications exist and some of the PETSc designers are strongly involved in the design and implementation of the MPI standard.

8.4 Bellman-Ford Graph Algorithm

In this section the performance of the Janus implementation of the Bellman-Ford shortest path algorithm is investigated. The Janus implementation of this algorithm has been discussed in Section 7.4.

On the one hand, the runtime of the routine in Figure 7.27 (configured for a sequential architecture) is compared with that of the `bellman_ford_shortest_paths` routine of the *Boost Graph Library*[67] (BGL). On the other hand, the speedup of a parallel configuration of the routine in Figure 7.27 is examined.

In both cases, the test graphs consist of triangulations, that is, the graph nodes are the vertices of of the triangles and the graph edges are the edges of the triangles. Figure 3.5 on Page 44 gives an examples of such a graph. The length of the edges are chosen as the graph weights. Thus, the Bellman-Ford shortest path algorithm literally finds the shortest path form on vertex of the triangulation to all others.

There is one important difference between the BGL and Janus implementation of Bellman-Ford. The `accumulator` object in Figure 7.27 is used to determine whether the algorithm can be terminated. The BGL implementation does not terminate early—for unknown reasons. In particular for sparse graphs, the savings in runtime can be tremendous. For the sake of fair comparison the `accumulator` is not used in the Janus implementation.

Table 8.6 shows timing results of the sequentially configured Janus routine and the BGL routine.

The ratio of the Janus and BGL implementations is approximately 0.45. This means that the sequential Janus is more than two times faster than its BGL counterpart. The performance advantage of Janus is caused by the

Nodes	Edges	Time BGL	Time Janus	Overhead Ratio
4,097	11,968	7.2	3.2	0.45
8,176	24,076	30.5	13.7	0.45
13,466	39,829	90.1	40.3	0.45

Table 8.6: Runtime results for sequential Janus and Boost implementations of Bellman-Ford

data structures used in the Janus implementation that are better suited for sparse structures such as finite element triangulations.

The Janus program uses the classes `jns::sorted_domain<int>` (§5.2.1.1) and `jns::relation` (§5.3.1.1) to represent the nodes and edges of the graph. Since the `jns::relation` class rests on the compressed row storage (CRS) scheme it is particular well suited for sparse graphs such as triangulations. The BGL example, on the other hand, deploys its general purpose `edge_list` data structure whose memory access patterns are for sparse graphs not as efficient as that of `jns::relation`.

Table 8.7 shows time measurements and relative speedup of the parallel Janus implementation when using only one processor per board. Up to 8 processors the speedup is super linear–due to fewer cache misses. However, this also shows that particular care has been taken to achieve an efficient implementation of the `accessor` methods.

Processors	1	2	4	8	16
Time	39.1	17.9	8.2	4.6	3.5
Relative Speedup	1.0	2.2	4.8	8.5	11.2

Table 8.7: Runtime and speedup for a parallel implementation of Bellman-Ford when using one processor per board

Table 8.8 shows time measurements and relative speedup of the parallel Janus implementation when using both processors on a board board. Again, the speedup is super linear up to 8 processors. However, not as good as in the case of using one processor per board. For 16 processor (on eight boards) there is only a speedup of 8.7 and for 32 processors there is even a slowdown. This is related to the local problem size that is getting too small to perform enough work between communication steps.

The data structures that have been used in the implementation of Bellman-Ford are the same as those used in the implementation of the fi-

Processors	1	2	4	8	16	32
Time	39.1	18.4	8.2	4.8	4.5	7.0
Speedup	1.0	2.1	4.8	8.1	8.7	5.6

Table 8.8: Runtime and speedup for a parallel implementation of Bellman-Ford when using both processors per board

nite element method in Section 7.2. Yet, in the case of Bellman-Ford the scalability is not as good as for the conjugate gradient method (see Section 8.3).

The main reason is that in the case of Bellman-Ford a much smaller triangulation has been used. It consists only of approximately 13,000 vertices and less than 40,000 edges. This graph has been chosen since without early termination the runtime of Bellman-Ford depends on the product of number of edges and number of nodes. The conjugate gradient method, on the other hand, performs usually much less iterations than there are nodes. Smaller problem size of the data-parallel part of the computation, however, means worse scalability.

8.5 Conclusions

The main result regarding the performance of Janus is that it favorably compares with state-of-the-art implementations in the field of data-parallel applications. This holds in particular for the scalability of Janus applications that have been configured to use MPI on a Linux cluster system. Thus, a major goal of the Janus effort has been achieved: Providing expressive and efficient parallel abstractions that do not rely on language extension or non-standard tools.

With respect to the solution of a finite element problem, Janus has been compared with the Portable Extensible Toolkit for Scientific computing (PETSc). The direct comparison of Janus with PETSc shows better results for the latter software package for a high number of processors. This is attributed to the more sophisticated utilization of MPI in PETSc.

The evaluation of the Janus implementation of *Game of Life* has shown that Janus components introduce almost no overhead compared to applications that have been completely written in the C programming language. This is due to the Janus implementation that uses compile time polymorphism rather than dynamic polymorphism. In other words, Janus uses class and function templates and avoids virtual functions for the implementation of its core components. Of course, the low overhead of the Janus components when compared with hand-written code is also due to the good optimization capabilities of state-of-the-art C++ compilers.

With regard to the Bellman-Ford algorithm, the Janus data structures are more efficient for sparse graphs than those of Boost Graph Library. However, it has to be taken into account that the Janus data structures have been specially designed for sparse relations whereas a general data structure of BGL was used.

Chapter 9

Discussion

This chapter concludes this dissertation by discussing the Janus framework and its components in different contexts. In particular, the influence of various software engineering paradigms on the design and implementation of Janus is investigated. Most notably in this context are *domain engineering*, *generic programming*, and *program families*. The highlights and limitations of Janus are compared with related frameworks in the field of scientific computing. This chapter also provides an outlook of upcoming research themes.

9.1 Janus in the Context of Domain Engineering

In the context of domain engineering a *domain* is considered as an area of knowledge that includes concepts and a terminology for that area but also knowledge of how to construct software systems in that area. Domain engineering attempts to take advantage of the knowledge and software products that are engineered in the domain. The emphasis is on knowledge management and engineering reusable software to accelerate the development of specific systems in the domain. A key activity of domain engineering is *domain analysis* that gathers essential information and formulates general requirements for systems in the domain.

The Janus framework has been designed for the domain of data-parallel applications. In Chapter 3, a thoroughly domain analysis has been undertaken by investigating a large range of data-parallel applications. This analysis started with investigating finite difference and cellular automata simulations that mostly rely on rectangular grids and stencil-like operations.

Finite element methods and adaptive mesh refinement algorithms have also carefully been surveyed. Finally, a typical parallel graph algorithm has been analyzed.

As a result of this analysis several general design guidelines have been formulated.

1. A clear classification of the objects that occur in data-parallel application has been carried out. Firstly, there are (finite) *sets* across which simultaneous operations are performed. Secondly, there are *relations* between these sets that describe data dependences. Finally, there are attributes or *properties* associated with the elements of the sets and relations. These attributes describe the data on which most of the parallel computations are performed.

2. Based on this classification rests the insight that the sets and relations are *more stable* than the data associated with them.

The clear separation into *spatial structures*, that is, the sets and relations, and associated data is usually not found in other approaches to develop software in the realm of scientific computing. State-of-the-art representatives of scientific software use vectors and matrices as basic abstraction. Examples are PETSc[6] and the Matrix Template Library[72]. In Janus, however, a matrix is considered as a pair of a relation and the coefficients that are associated with the relation.

One advantage of this clear separation is that it simplifies the use of different implementation techniques and data structures for sets and relations on the one hand, and associated data on the other hand. Deploying different data structures is recommendable because of the relative stability of spatial structures.

The stability aspect of the spatial structures also puts into perspective the necessity of dynamic data structures such as trees or lists. Rather it is sufficient to use less flexible yet more efficient data structures as they occur in the context of sparse matrices.

Emphasizing the relative stability is, nevertheless, a restriction on the dynamics of data-parallel problems. In particular, the design of Janus precludes highly dynamic particle simulation methods.

9.2 Janus in the Context of Generic Programming

In domain engineering, the steps that follow the analysis are *domain design* and *domain implementation.* The objective of domain design is the development of a common architecture for the systems in the domain.

The ideas of *generic programming* have been used to define the Janus architecture.

Generic programming has become famous through the Standard Template Library—a framework of containers, iterators, and algorithms. The components of STL—and even more so the conceptual framework behind them—proved the expressiveness, usability, high performance, and extendibility for the domain of fundamental data structures and algorithms of computer science. This promising approach has been applied by researchers to other fields, for example, matrix computations[72] and graph algorithms[67]. Not surprisingly, these are domains where performance is a major design concern.

The conceptual framework of Janus rests on that of STL and introduces new concepts that reflect the particular requirements of data parallel scientific applications.

The three major concepts of Janus are Domain, Relation, and Property Function. They formulate syntactic and semantic requirements for the representation of (distributed) sets, their relations, and data that are associated with them. The most important feature is the explicitly stated *relative stability* of Domain and Relation with respect to Property Function. At the same time the concept of relative stability is flexible enough to represent irregular and dynamically created domains and relations.

This flexibility is more precisely specified in the concepts of One-Phase Domain and Two-Phase Domain. One-phase structures are used, for example, to represent rectangular grids. Two-phase structures, on the other hand, are deployed to represent irregular or dynamic problems. Thus, the Janus conceptual framework is flexible enough to support a wide range of data-parallel problems.

Two-phase structures also simplify the problem of hiding the details of constructing distributed data structures. Many of the Janus algorithms, for example *accumulators*, have this clear separation into *initiation* and *completion* phases.

Generic programming frameworks are often implemented as C++

libraries—and Janus is no exception. Using C++ has the advantage that it heavily relies on compile-time polymorphism. As a result, generic components can be as efficient as hand-written components. However, a drawback is that generic components cannot be easily exchanged at run time. Dynamic polymorphism provides here more flexibility.

9.3 Janus as a Program Family

Domain engineering aims at developing reusable software components to accelerate the construction of systems in a domain. This is very closely related to Parnas' idea of program families [84]. A more recent extension of this idea is *software product line engineering* [107].

Janus is in many aspects a program family. First of all, its components are C++ template classes and template functions that yield a family of concrete data structures and algorithms when instantiated with built-in or user-defined types. Though this seems a trivial aspect, it is worth emphasizing that a state-of-the-art scientific framework such as PETSc (written in the C programming language) has much greater difficulties to integrated user-defined types.

Secondly, the conceptual framework of Janus allows to define components, that is *family members*, that are highly customized and efficient and not impaired by too general functionalities. The `grid` and `stencil` template classes are very lean and efficient data structures for data-parallel problems on rectangular grids.

Another example for the occurrence of sub-families within Janus are the different platforms for which the components can be configured. In particular, distributed-memory hardware architectures can be efficiently utilized on top of MPI. However, the components can also be configured to run efficiently on a single-processor architecture.

The main reason for the ability to efficiently utilize different computing platforms is that the Janus concepts are very application-oriented. In particular, the two-phase data structures and algorithms and the data transfer primitives of the Relation concept make it very easy to hide platform dependent issues such as buffering of data that shall be sent to a remote address space.

At the same time, it must be noted that the fundamental issues of distributed-memory architectures are already addressed by the concepts of

Janus. Most notably are here the concepts Distributed Domain and Distributed Relation. Moreover, the process model of Janus follows this of MPI. A drawback of this predisposition towards distributed-memory architectures is that shared-memory parallelism can currently not as easily exploited by Janus.

9.4 Perspectives

As pointed out in this dissertation, Janus is well suited for a broad range of data-parallel applications. However, highly dynamic particle simulation methods can not represented within the Janus framework very well currently. The Janus framework should be extended to be able to cope with these issues.

Deepening the domain analysis of data-parallel applications, Janus modules that provide service for particular sub-domains should be developed. This holds in particular for mesh-restructuring methods that despite the diversity of element shapes, restructuring rules, or remapping strategies expose a lot of commonalities. A closer examination of the relationships between the Janus concepts and the ideas of (distributed) data bases could further improve the usability and interoperability of Janus.

The very important domain of embedded and real-time software systems also shares considerable commonalities with the domain of scientific computing. In particular, the real-time constraints and the limited computing resources preclude many standard software-engineering techniques and platforms. Here again, lean and application-oriented abstractions are the key to achieve reuse. Similar to Janus in the field of data-parallel scientific applications, generic programming can deliver flexible and efficient work products here as well.

Bibliography

[1] Jason Abate, Peng Wang, and Kamy Sepehrnoori. Parallel compositional reservoir simulation on clusters of PCs. *Int. J. High Performance Computing Applications*, 15(1):13–21, Spring 2001.

[2] Mark F. Adams. Evaluation of three unstructured multigrid methods on 3d finite element problems in solid mechanics. *SIAM J. Sci. Comput.*

[3] S.S. Alhir. *UML in a Nutshell.* O'Reilly, 1998.

[4] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High Performance Scientific Computing. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, August 1999.

[5] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Third Edition.* Addison-Wesley, 2000.

[6] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries Programming. In A. M. Bruaset E. Arge and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[7] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc Web page. http://www.mcs.anl.gov/petsc.

[8] J.E. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324:446–449, 1986.

[9] J.J. Barton and L.R. Nackman. *Scientific and Engineering C++.* Addison-Wesley, 1994.

[10] P. Beckman. Home Page of PAWS: Parallel Application WorkSpace. `http://www.acl.lanl.gov/paws`.

[11] P.H. Beckman, P.K. Facel, W.F. Humphrey, and S.M. Mniszewski. Efficient Coupling of Parallel Applications Using Paws. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[12] M. Besch and H.W. Pohl. Topographic Data Mapping by Balanced Hypersphere Tessellation. In *Proceedings of Euro-Par Conference*, LNCS, Lyon, France, August 1996. Springer-Verlag.

[13] A. Bik and H. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, 1993.

[14] George Biros and Omar Ghattas. A lagrange-newton-krylov-schur method for pde constrained optimization. *SIAG-Opt Views and News*, 11(2), august 2000.

[15] R. Biswas and R.C. Strawn. Tetrahedral and hexahedral mesh adaption for CFD problems. *Appl. Num. Math.*, 20:337–348, 1996.

[16] R. Biswas and R.C. Strawn. Mesh quality control for multiply-refined tetrahedral grids. Technical Report NAS-97-007, NASA, 1997.

[17] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995. `www.myricom.com`.

[18] Per Bothner and et.al. *Home Page of Gnu Compiler Collection.* Free Software Foundation. `http://gcc.gnu.org`.

[19] D. Braess. *Finite Elemente.* Springer, 1994.

[20] D.L. Brown, W.D. Henshaw, and D.J. Quinlan. Home Page of Overture: Object-Oriented Tools for Solving CFD and Combustion Problems in Complex Moving Geometry. `http://www.llnl.gov/CASC/Overture/`.

[21] D.L. Brown, W.D. Henshaw, and D.J. Quinlan. Overture: An Object-Oriented Framework for Solving Partial Differential Equations. In *Proceedings of ISCOPE 1997*, volume 1343 of *LNCS*, pages 177–184, Marina del Rey, California, USA, December 1997. Springer-Verlag.

[22] K. R. Buschelman, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc and overture: Lessons learned developing an interface between components. Technical Report ANL/MCS-P858-1100, Argonne National Laboratory, 2000. to appear in the Proceedings of the International Federation for Information Processing Working Conference on Software Architectures for Scientific Computing, Kluwer.

[23] C. Campbell and D. Redfern. *The MATLAB 5 Handbook*. 1998.

[24] J. O. Coplien. *Multi-paradigm design for C++*. Addison Wessley Longman, Inc., 1998.

[25] T.H. Cormen, Leiserson C.E., and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1998.

[26] K. Czarnecki and Eisenecker U.W. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[27] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in p3l. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of EURO-PAR '97, Passau, Germany*, volume 1300 of *LNCS*, pages 619–628. Springer, August 1997.

[28] B. Dawes. *Home Page of C++ Boost*. C++ Boost. `http://www.boost.org`.

[29] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[30] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. IML++ v. 1.2: Iterative Methods Library Reference Guide. Technical Report UT, CS-95-303, National Institute of Standards and Technology and University of Tennessee, Knoxville, August 1995. available via `http://www.netlib.org/lapack/lawns/lawn102.ps`.

[31] J. J. Dongarra, J.D. Cross, and R.J. Hammarling, S. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[32] P. Enskonatus and M. Kessler. Concepts and Formal Description of the PROMOTER Language, Version 2.0. Technical report, GMD FIRST, 1997. RWC-TR-96-020, `http://www.first.gmd.de/promoter/papers/`.

[33] D. Flangan, J. Farley, and W. Crawford. *Java Enterprise in a Nutshell*. O'Reilly & Associates Inc., 1999.

[34] Matthew Flatt. Programming languages for reusable software components. Technical Report TR99-345, 20, 1999.

[35] CCA Forum. Home Page of the Common Component Architecture Forum. `http://www.acl.lanl.gov/cca-forum/`.

[36] I. Foster and C Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[38] Dennis Gannon. *Home Page of the Common Component Architecture Forum*. Distributed Computing Research Team of Sandia National Laboratories. `http://z.ca.sandia.gov/~cca-forum/`.

[39] J. Gerlach. PROMOTER—First Application Study, Navier-Stokes Equations. Technical Report TR-94-018, Real World Computing Partnership, Japan, 1994.

[40] J. Gerlach, P. Gottschling, and H.W. Pohl. Core Components of Janus, Release 2.0. Technical Report TR D-00-028, Real World Computing Partnership, Japan, 2000.

[41] J. Gerlach, G. Heber, and A. Schramm. Finite Element Methods in the PROMOTER Programming Model. In *Proceedings of the International EUROSIM Conference on HPCN Challenges*, Delft, Netherlands, June 1996.

[42] J. Gerlach, Z.Y. Jiang, and H.P. Pohl. Integrating OpenMP into a Generic Framework for Parallel Scientific Computing. In *Workshop on OpenMP Applications and Tools*, Lecture Notes in Computer Science, Purdue University, West Lafayette, Indiana, USA, July 2001. Springer-Verlag.

[43] J. Gerlach and M. Sato. Generic Programming for Parallel Mesh Problems. In *accepted for the Third International Symposium of Computing in Object-Oriented Parallel Environments ISCOPE 99*, San Francisco, USA, December 1999.

[44] J. Gerlach, M. Sato, and Y. Ishikawa. Janus: A C++ Template Library for Parallel Dynamic Mesh Applications. In *Proceedings of the Second International Symposium of Computing in Object-Oriented Parallel Environments ISCOPE 1998*, volume 1505 of *Lecture Notes in Computer*

Science, pages 215–222, Santa Fe, New Mexico, USA, December 1998. Springer-Verlag.

[45] W.K. Giloi, M. Kessler, and A Schramm. PROMOTER : A High Level Object-Parallel Programming Language. In *Proceedings of International Conference on High Performance Computing*, New Delhi, India, December 1995.

[46] W.K. Giloi, M. Kessler, and A. Schramm. A High-Level, Massively Parallel Programming Environment and Its Realization. Technical Report TR-96-001, Real World Computing Partnership, Japan, 1996.

[47] Java Grande. Home Page of the Java Grande Forum. `http://www.javagrande.org`.

[48] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[49] Object Management Group. *Home Page of the Common Object Request Broker Architecture.* Object Management Group. `http://www.corba.org`.

[50] F. Guidec and J.-M. Jézéquel. Design of a Parallel Object-Oriented Linear Algebra Library. In *Programming Environment for Massively Parallel Distributed Systems, Proceedings of the IFIP Working Conference WG 10.3*, pages 359–364, April 1994.

[51] A. N. Habermann, Peter Feiler, Lawrence Flon, Loretta Guriano, Lee Cooprider, and Bob Schwanke. Modularization and Hierarchy in a Family of Operating Systems. *Communications of ACM*, 5(19):266–272, 1976.

[52] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations.* Springer, 1994.

[53] G. Heber. QCD algorithms in the Promoter programming model. Technical Report TR TR-95-058, Real World Computing Partnership, Japan, 1995.

[54] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[55] R.D. Hornung and S.R. Kohn. The Use of Object-Oriented Design Patterns in the SAMRAI Structured AMR Framework. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*. SIAM, October 1998.

[56] W. Humphrey, S. Karmesin, F. Bassetti, and J. Reynders. Optimization of Data-Parallel Field Expressions in the POOMA Framework. In *Proceedings of ISCOPE 1997*, volume 1343 of *LNCS*, pages 185–194, Marina del Rey, California, USA, December 1997. Springer-Verlag.

[57] J. Irwing, J.-M. Loingtier, J.R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-Oriented Programming Sparse Matrix Code. In *Proceedings of ISCOPE 1997*, volume 1343 of *LNCS*, pages 249–256, Marina del Rey, California, USA, December 1997. Springer-Verlag.

[58] Yasuo Ishii. *Introduction to SoftwareEngieering (in Japanese)*. Nikka-Giren Publishing, fourth edition, 1991.

[59] Yutaka Ishikawa, Hiroshi Tezuka, Atsuhi Hori, Shinji Sumimoto, Toshiyuki Takahashi, Francis O'Carroll, and Hiroshi Harada. RWC PC Cluster II and SCore Cluster System Software – High Performance Linux Cluster. In *Proceedings of the 5th Annual Linux Expo*, pages 55–62, 1999.

[60] E. Johnson and D. Gannon. Programming with the HPC++ Parallel Standard Template Library. In *SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, Minnesota, March 1997.

[61] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T.J. Williams. Array Design and Expression Evaluation in POOMA II. In *Proceedings of ISCOPE 1998*, volume 1505 of *LNCS*, pages 231–238, Santa Fe, New Mexico, USA, December 1998. Springer-Verlag.

[62] Steve Karmesin. Home Page of POOMA: Parallel Object-Oriented Methods and Applications. `http://www.acl.lanl.gov/Pooma/index.html`.

[63] G. Karypis. *Metis a Family of Multilevel Partitioning Algorithms*. http://www-users.cs.umn.edu/˜karypis/metis/main.shtml.

[64] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwing. Aspect-Oriented Programming. In

Proceedings of European Conference on Object-Oriented Programming (ECOOP), volume 1241 of *LNCS*, Finland, June 1997. Springer-Verlag.

[65] S. Kohn. *Home Page of SAMRAI: Structured Adaptive Mesh Refinement Applications Infrastructure.* Lawrence Livermore National Laboratory.
http://www.llnl.gov/CASC/SAMRAI.html.

[66] Kuck & Associates. *Home Page KAI.*
`http://www.kai.com`.

[67] L.Q. Lee, J. Siek, and A. Lumsdaine. *Home Page of the Boost Graph Library (BGL).* C++ Boost. `http://www.boost.org/libs/graph/doc/table_of_contents.html`.

[68] L.Q. Lee, J. Siek, and A. Lumsdaine. *Home Page of the Generic Graph Component Library (GGCL).* Laboratory for Scientific Computing, Department of Computer Science and Engineering University of Notre Dame. `http://www.lsc.nd.edu/research/ggcl`.

[69] L.Q. Lee, J. Siek, and A. Lumsdaine. Generic Graph Algorithms for Sparse Matrix Ordering. In *Proc. of ISCOPE 99*, LNCS, San Francisco, December 1999. Springer-Verlag.

[70] B. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5), May 1988.

[71] Andrew Lumsdaine, Lie-Quan Lee, and Jeremy Siek. Home Page of the Iterative Template Library (ITL). `http://www.lsc.nd.edu/research/itl`.

[72] Andrew Lumsdaine and Jeremy Siek. Home Page of the Matrix Template Library (MTL). `http://www.lsc.nd.edu/research/mtl`.

[73] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar. Next Generation Generic Programming and its Application to Sparse Matrix Computations. In *Proceedings of International Conference on Supercomputing, 2000*, 2000.

[74] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[75] Microsoft. *Home Page of Microsoft's Component Object Model (COM).* Microsoft. `http://www.microsoft.com/com`.

[76] Sun Microsystems. *Home Page of JavaBeans Component Architecture.* Sun Microsystems. `http://java.sun.com/products/javabeans/`.

[77] D. R. Musser. A Generic Programming Concept Web. `http://www.cs.rpi.edu/~musser/concept-web/index.html`.

[78] D. R. Musser. Generic Programming. `http://www.cs.rpi.edu/~musser/gp/index.html`.

[79] D.R. Musser and A.A. Stepanov. Generic Programming. In *First Int. Joint Conf. of ISSAC-88 and AAECC-6*, volume 358 of *LNCS*, pages 13–25. Springer, June 1988.

[80] Netlib Repository at UTK and ORNL. *Document for the Basic Linear Algebra Subprograms (BLAS) Standard.* working document of the BLAST Sparse Subcommittee, available from `http://www.netlib.org/utk/papers/sparse.ps`.

[81] C++ Resources Network. *Home Page of the C++ Resources Network.* C++ Resources Network. `http://www.cplusplus.com`.

[82] Object-Oriented Numerics. Home Page of Scientific Computing in Object-Oriented Languages. `http://www.oonumerics.org`.

[83] S. J. Owen. *Webpage of Finite Element Mesh Generation.* `http://www.andrew.cmu.edu/user/sowen/mesh.html`.

[84] D. L. Parnas. On the Design and Development of Program Families. In *IEEE Transactions on Software Engineering*, pages 1–9. IEEE, 1976.

[85] D.l. Parnas. Designing software for ease of extension and contraction. *IEEE Transaction on Software Engineering*, SE-5(2), mar 1979.

[86] The Globus Project. Home Page of Nexus: The Nexus Multithreaded Communication Library. `http://www.globus.org/nexus/`.

[87] The Globus Project. Home Page of the Globus Project. `http://www.globus.org`.

[88] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys, September 1984*, 16(3):319–348, 1984.

[89] RWCP Parallel and Distributed Systems GMD Laboratory. *Home Page of the PROMISE Project.* `http://www.first.gmd.de/promise`.

[90] D. Schmidt, M. Stal, H. Rohnert, and Buschmann F. *Pattern-oriented Software Architecture Vol 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[91] Douglas C. Schmidt. *Users Guide to gperf*. Free Software Foundation. `http://www.gnu.org/manual/gperf-2.7/gperf.html`.

[92] A. Schramm. Irregular Applications in PROMOTER. In Shriver B. Giloi W.K., Jhnichen S., editor, *Proceedings of the Internatinal Conference on Massively Parallel Programming Models*. IEEE-CS Press, October 1995.

[93] J. R. Shewchuk. *A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*. http://www.cs.cmu.edu/˜quake/triangle.html.

[94] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *First Workshop on Applied Computational Geometry*, pages 124–133. ACM, May 1996.

[95] J. G. Siek and A. Lumsdaine. The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Algebra. In *Proceedings of ISCOPE 1998*, volume 1505 of *LNCS*, pages 59–70, Santa Fe, New Mexico, USA, December 1998. Springer-Verlag.

[96] B. Smith. Personal Communication. Nov. 17, 2000.

[97] M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, and J. Dongarra. *MPI—The Complete Reference, Volume 1, The MPI Core*. The MIT Press, 1998.

[98] Alex Stepanov. Standard Template Library Programmer's Guide. http://www.sgi.com/Technology/STL.

[99] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1998.

[100] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[101] The International Standards Organization (ISO). *Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++*, March 1997. ftp://ftp.research.att.com/dist/c++std/WP/CD2/.

[102] The OpenMP Consortium, www.openmp.org. *OpenMP: Draft C/C++ 2.0 specification, with change bars*, November 2001. http://www.openmp.org/specs/mp-documents/draft_cspec20_bars.pdf.

[103] M. Tief and J. Gerlach. High-Quality Geometric Repartitioning with Mosaik. In *Proceedings of The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, June 2001. CSREA Press.

[104] U. Trottenberg, C.W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.

[105] T.L Veldhuizen. Arrays in Blitz++. In *Proceedings of the Second International Symposium of Computing in Object-Oriented Parallel Environments ISCOPE 1998*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230, Santa Fe, New Mexico, USA, December 1998. Springer-Verlag.

[106] P. Wegner. Classification in object-oriented systems. *SIGPLAN*, 21(10):173–173, October 1986.

[107] M.W. Weiss and R.L.. Tau. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[108] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. Technical Report UT, CS-00-448, National Institute of Standards and Technology and University of Tennessee, Knoxville, September 2000.

[109] Olof B. Widlund. Schwarz methods for Helmholtz's equation. In John A. DeSanto, editor, *Mathematical and Numerical Aspects of Wave Propagation*, pages 620–622, Philadelphia, 1998. SIAM. Proceedings of the Fourth International Conference on Mathematical and Numerical Aspects of Wave Propagation, Golden, Colorado, June 1–5.

Lebenslauf

Name:	Jens Gerlach
Anschrift:	Wolliner Straße 69, 10435 Berlin
Geburtsdatum:	8. August 1965
Geburtsort:	Jena
Staatsbürgerschaft:	Deutsch
Familienstand:	ledig
Kinder:	eine Tochter

Bildungsweg

1972–1981	Polytechnische Oberschule
1981–1984	Erweiterte Oberschule
1984	Abitur
1984–1986	Grundwehrdienst
1986–1991	Studium der Mathematik an der Humboldt-Universität zu Berlin, Abschluss als Diplom-Mathematiker

Bisherige Arbeitsverhältnisse

1992–1996	Wissenschaftlicher Mitarbeiter bei GMD-FIRST Projekte: DYMOS und PROMOTER
1996–1999	Senior Researcher at Tsukuba Research Center of Real World Computing Partnership, Japan
1999–2002	Wissenschaftlicher Mitarbeiter bei Fraunhofer-FIRST Promise Projekt, Projektleiter EMPRESS