Distribution and Parallelization Strategies for Integrated Simulation, Visualization and Rendering Systems

---

Verteilungs- und Parallelisierungsstrategien für integrierte Simulations-, Visualisierungs- und Renderingsysteme

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Peter Kipfer

Erlangen – November 2002

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	22. November 2002
Tag der Promotion:	18. Februar 2003
Dekan:	Prof. Dr. A. Winnacker
Berichterstatter:	Prof. Dr. G. Greiner
	Prof. Dr. U. Rüde

Abstract

In nearly all scientific disciplines, numerical simulation plays an important role for verifying design constraints, performing quantitative and qualitative measurements and asking what-if questions. The basis for successful implementation is contributed by well understood numerics research in computer science. This has enabled the development of highly accurate simulation codes with predictable error bounds.

Nowadays concrete numerical simulation problems produce enormous amounts of result data because of the continuous increase in computing power of large scale computing facilities. The results must be processed with scientific methods for visual display, in order to allow easy interpretation. The resources needed for these methods surpass the capabilities of current generation desktop systems by far. Additionally, the comparably low bandwidth of the I/O channels of supercomputers strongly suggest to perform the post-processing on the large machine.

Additionally, nowadays available computer architectures provide a large variety of special purpose hardware. Many of these features have been introduced to deal with common bottlenecks. High-performance applications therefore have to employ these features in order to make full use of the processing potential of current hardware. Without using these features, the application will not be able to utilize the gain in processing speed of the next hardware generation.

This thesis examines several principal strategies for handling distribution and parallelization. A classification is worked out to demarcate the areas of application. Using these strategies, a library for integrated simulation, visualization and rendering on supercomputers and desktop systems is implemented. It offers a clear separation of functionality by employing several abstraction levels to the programmer. Therefore, this thesis presents a substantial contribution to scientific computing to cope with grand-challenge problems. It supplies the enabling key technology for efficient post-processing for visualization and rendering on the supercomputer.

Organization The thesis is split into three major parts. The first part presents a classification of distribution and parallelization strategies and discusses their properties. In the second part, the *gridlib* framework library is presented. It has been developed in

the context of this thesis to evaluate the strategies for building a software infrastructure for integrated simulation, visualization and rendering. Detailed descriptions of applications using the *gridlib* library are presented in the third part of the thesis. They deal with several specific aspects of nowadays computer architectures and provide efficient solutions for high performance applications.

After a summary of the results of this thesis, the appendix lists the most important grid management interfaces of the *gridlib* library. An additional glossary explains technical terms and abbreviations. The three main parts feature margin labels to draw the attention of the reader to miscellaneous aspects:

Technical terms and abbreviations within this paragraph are explained in the glossary section. There are also references to explanatory notes within the text.

This list assembles the most important facts that have to be taken into account when implementing an application.

This presents a discussion of the pros and cons of the strategy treated. They are the most important aspects when considering the usage of the strategy within an application design.

Revision 1.4

Contents

Abstract 3

List of Figures 12

List of Tables 13

List of Program Listings 15

List of Color Plates 17

Acknowledgements 19

I PARALLELIZATION AND DISTRIBUTION STRATEGIES

1 Coarse strategies 27
- 1.1 Introduction 27
- 1.2 Message passing systems 28
  - 1.2.1 PVM 28
  - 1.2.2 MPI 29
- 1.3 Distributed object-oriented systems 32
  - 1.3.1 ACE 32
  - 1.3.2 CORBA 34
  - 1.3.3 TAO 36
  - 1.3.4 The POA object adapter 38
- 1.4 Discussion 40

2 Intermediate strategies 43
- 2.1 Introduction 43
- 2.2 Pipelining 44
- 2.3 OpenMP multi-threading 45
- 2.4 Discussion 47

3 Fine strategies 51
- 3.1 Introduction 51
- 3.2 SIMD processing 52
- 3.3 Custom hardware programming and design 55
- 3.4 Discussion 58

II INTEGRATING SIMULATION, VISUALIZATION AND RENDERING

4 The *gridlib* project 65
4.1 Introduction 65
4.2 Overview 66
4.3 Storage abstraction layer 67
4.3.1 Memory Pools 67
4.4 Element abstraction layer 68
4.4.1 External Polymorphism 68
4.5 Mesh abstraction layer 72
4.5.1 Algorithmic abstraction 72
4.6 Clients 74
4.6.1 Services and Utilities 74
4.6.2 Visualization and Rendering 75

5 *gridlib* applications 79
5.1 Evaluating the quality of tetrahedral grids 79
5.2 Progressive isosurfaces from tetrahedral grids 80
5.3 Fast time-dependent isosurfaces 80
5.4 Visualization across partitions 81
5.5 Mesh registration 81

III APPLIED PARALLELIZATION AND DISTRIBUTION

6 Simulation 85
6.1 SIMD processing for Lattice Boltzmann methods 85
6.1.1 Lattice gas 85
6.1.2 Lattice Boltzmann 86
6.1.3 Driven cavity simulation 87

7 Visualization 93
7.1 Interactive display of time dependent volumes 93
7.1.1 Displaying scalar volumes 93
7.1.2 Displaying vector volumes 98
7.2 Local exact particle tracing 99
7.2.1 Integration methods 100
7.2.2 Cell classification 102
7.2.3 Parallel pre-processing 102
7.2.4 Building a smooth curve 104

8 Rendering 107
8.1 Transparent Distributed Processing For Rendering 107
8.1.1 Distributed Lighting Networks 108
8.1.2 Design Patterns for Transparent Distribution 108

8.1.3 Discussion 113
8.2 Parallel rendering 116
8.2.1 Rasterizer performance 117
8.2.2 Optimization 120
8.3 Ray tracing in hardware 121
8.3.1 System Overview 122
8.3.2 Implementation 126
8.3.3 Results 127

IV CONCLUSION

9 Summary 135

10 Future Challenges 139
10.1 Integration of functionality 139
10.2 Flexible SIMD processing 139
10.3 Integrated FPGA technology 140

V APPENDIX

A Glossary 145

B Color Plates 151

C Interface inheritance in *gridlib* 159

D The *gridlib* mesh interface 163

Bibliography 165

Index 170

List of Figures

1 Parallelization effort . . . 24
2 Classification of parallel architectures . . . 25
3 Amdahl's law of scalability . . . 26
1.1 The PVM master–slave programming model . . . 28
1.2 Buffered point-to-point communication in MPI . . . 30
1.3 Derived data types in MPI . . . 31
1.4 Architecture of the ACE framework . . . 33
1.5 The CORBA ORB architecture . . . 35
1.6 The TAO architecture . . . 37
1.7 Using the Portable Object Adapter . . . 39
2.1 A pipeline example . . . 44
3.1 Using SIMD instructions . . . 52
3.2 Computing the dot product with SIMD . . . 53
3.3 Computing the square root with SIMD . . . 54
3.4 *n*VIDIA register combiners . . . 56
3.5 Internal layout of the Virtex FPGA . . . 57
3.6 Adding two integer vectors in hardware . . . 58
4.1 The *gridlib* architecture overview . . . 66
4.2 The external polymorphism design pattern . . . 69
4.3 Arranging the storage of primitive objects . . . 70
4.4 Pseudo–polymorph call execution . . . 71
4.5 Renderer hierarchy . . . 76
6.1 Lattice gas . . . 86
6.2 Lattice Boltzmann . . . 87
6.3 Driven cavity flow . . . 88
6.4 Performance of the SIMD Lattice Boltzmann solver . . . 89
6.5 Cache optimized driven cavity simulation . . . 90
7.1 Tiling of volume slices . . . 94
7.2 The stream processing pipeline . . . 95
7.3 Multi-pass rendering of large volumes . . . 96
7.4 Time dependent scalar volumes . . . 97
7.5 Processing time dependent volumes . . . 98
7.6 Time dependent vector volumes . . . 99

7.7 Building a smooth particle curve . . . 104
7.8 High-quality visualization for particle tracing . . . 105
7.9 Local exact particle tracing . . . 105
8.1 Wrapping existing computational objects . . . 109
8.2 Multiplexers for parallelization . . . 110
8.3 Services for distributed LightOps . . . 112
8.4 Data flow for parallel rendering . . . 117
8.5 The distributed framebuffer . . . 118
8.6 Scaling of parallel rendering . . . 119
8.7 Efficiency of parallel rendering . . . 119
8.8 Saturation of communication channels . . . 120
8.9 The circuit layout of the FPGA ray tracer . . . 123
8.10 Intersection and normal calculation modules . . . 125
8.11 The FPGA development board . . . 126
8.13 Scenes rendered with the FPGA ray tracer . . . 128
8.12 The quadric test scene . . . 129
C.1 *gridlib* vertex inheritance diagram . . . 159
C.2 *gridlib* edge inheritance diagram . . . 160
C.3 *gridlib* geometry element inheritance diagram . . . 160
C.4 *gridlib* edge collaboration graph . . . 161
C.5 *gridlib* vertex collaboration graph . . . 162
C.6 *gridlib* geometry element collaboration graph . . . 162

List of Tables

1.1 Comparison of distribution and parallelization concepts 41
2.1 Comparison of modular program design and pipelining 48
4.1 Visualization algorithms of the *gridlib* 78
7.1 Comparison of integration methods 101
7.2 Extraordinary cells in local exact particle tracing 102
8.1 Efficiency of asynchronous data transfer 112
8.2 Using multiplexers for data-parallel tasks 115
8.3 Performance of a distributed lighting network 116
B.1 Visualization algorithms of the *gridlib* 157

List of Program Listings

1.1 Typical MPI programming pattern 29
2.1 Using OpenMP for shared memory multi-threading 46
3.1 Computing the dot product with 3DNow! 60
3.2 Assembler code for integer vector addition 61
4.1 Algorithmic abstraction on the mesh level 77
6.1 Lattice Boltzmann algorithm for driven cavity flow 87
6.2 Comparison of standard to SIMD-enabled programming 91
7.1 Pre-processing for local exact particle tracing 103
8.1 A multiplexer for lighting computations 111
8.2 The FPGA trace and shade algorithm 130
8.3 FPGA intersection module pseudocode 131

List of Color Plates

B.1 Drawing modes of the rendering subsystem 151
B.2 Visualization examples 152
B.3 Using the FPGA ray tracer 153
B.4 Properties of the FPGA ray tracer 153
B.5 A distributed lighting network 154
B.6 High-quality visualization for particle tracing 155
B.7 Progressive transmission and visualization 155
B.8 Local exact particle tracing 156
B.9 Evaluating the quality of tetrahedra 156
B.10 Progressive isosurface visualization 158
B.11 Fast isosurface extraction across partitions 158

Acknowledgements

First of all, I like to thank my supervisor Prof. Günther Greiner for his incredible support of my work, his friendship and constant willingness to explain complex mathematical topics in an understandable way to computer graphics people like me. Working with him has always been a pleasure both on the professional and human side. For inspiring discussions and valuable advice, I am grateful to Prof. Ulrich Rüde. He focused my interest on flow simulation and supported me through all the stages of this thesis.

I would like to extend a very special thank to Prof. Hans-Peter Seidel, Prof. Thomas Ertl and Prof. Philipp Slusallek who got me involved with computer graphics during my studies and provided an initial place of employment. Thank you for the good start !

Regarding the development of the *gridlib* library, I have to thank several people for supporting it. Especially Ulf Labsik, who helped to implement the basic meshing functionality, Frank Reck and Stefan Meinlschmidt, who pushed the visualization part and Dr. Christof Rezk-Salama for giving support with direct volume rendering techniques.

Research related to the *gridlib* development has also been performed and supported by the gifted team at the system simulation chair, namely Dr. Frank Hülsemann, Ben Bergen and Thomas Pohl. They provided valuable insights in numerical issues and voluntarily hosted my advanced C++ programming course.

This thesis would not have been possible with the support of all these people. Especially, I like to thank Prof. Marc Stamminger for proof-reading it and for sharing his profound knowledge on rendering topics.

Over all these years, the work would have been boring without all the people of the computer graphics team. Instead, it was real fun and I am sure I will miss our secretary Maria Baroti, the researchers Michael Bauer, Michael Scheuering, Grzegorz Soza, Fernando Vega and Gerd Sussner. You always made me look at the bright side of graphics. Especially, I like to thank Roman Sturm, who knows everything about Macs and videos, and my office mate Christian Vogelgsang, who speaks C++ more fluently than German, for their friendship and discussions.

Finally, and most of all, I like to thank my parents for supporting me both financially and ideologically, and my sister Vera and her husband Roland for spending so much time with boring computer science guys. My franconian roots have always been reinforced by the fabulous musicians of *Die ZEIDLER*, Theatermusik Hüttenbach and Stadtmusik Hersbruck and ensured me that there is more to life than computer graphics.

Peter Kipfer

Part I

Parallelization and Distribution Strategies

Although computational and memory resources have increased tremendously over the past years, solving *grand challenge* scientific problems still isn't possible on desktop or workgroup size machines. Some specialized areas already managed to break Moore's law[1], but it is doubtful whether this will become the regular case for all hardware components. The available resources of large scale computing facilities, i.e. *supercomputers* therefore must be orchestrated carefully to work together. Strategies for resource handling must be developed, that help to structure the problem and provide the programmer with complexity warranties for communication bandwidth, latency, scaling, bridgeable distance and others.

Parallelization and distribution strategies require a good understanding of the target system architecture. Because of the enormous differences in resource access speed, successful creation of a parallel or distributed application must honor the peculiarities of the target system. [Sch01a] has examined them for building an automatic load-balancing framework for Java programs. In the following, we briefly review architectural classifications and programming models, in order to give a short overview of parallel and distributed computing resources that have been examined in this thesis.

Flynn [Fly72] presented one of the most commonly used classifications of parallel architectures by examining dependencies between processor instructions and data streams:

- **single instruction, single data (SISD)** is the classical von Neumann architecture that does not exhibit parallelism.
- **single instruction, multiple data (SIMD)** systems are array or vector computers. Every instruction is executed on multiple data simultaneously.
- **multiple instruction, single data (MISD)** machines have not been built yet, because multiple instructions would work on the same data simultaneously, which is very questionable.
- **multiple instruction, multiple data (MIMD)** systems have multiple processors working independently on multiple data. This is the most flexible architecture.

G page 145

MIMD systems can be distinguished further by the memory each processor can access directly. A workstation cluster therefore is classified as *no remote memory access (NORMA)* architecture, while a Cray T90 system is classified as a *uniform memory access (UMA)* system. Because the cluster architecture demands more programming effort and the latter architecture is very expensive, intermediate solutions have been built that introduce distributed memory resources that are accessible uniformly with additional hardware support. Because local memory has lower latency, these architectures are classified as *non uniform memory access (NUMA)*. The NUMA concept however introduces another dependency: In order to accelerate memory access, every processor maintains a *cache* that will become inconsistent. Here, the programmer has to ensure, that he never accesses out-of-date data. Therefore, a solution has been proposed using

[1] *n*VIDIA claims to double rendering performance in 8 months (Moore: 18 months) [NVa]

additional hardware that automatically keeps the caches in sync. The SGI Origin system is one of these architectures classified as *cache coherent NUMA (ccNUMA)*.

Typically, parallelization and distribution strategies rely on programmer and compiler working together. Depending on the abstraction level, the compiler can perform automatic parallelization and hide details from the programmer (→ figure 1). However, subsequent parallelization or vectorization of existing implementations is nearly impossible if they have been written without paying attention to specific issues of parallel computation. Therefore the compiler often has only very limited success in performing the task automatically. Manual intervention is indispensable even on the lowest level. Consequently, it is common to rewrite critical sections of an algorithm for parallelization and distribution. This thesis investigates the existing possibilities for creating well structured distributed and parallel programs right from the start that are highly maintainable and efficient.

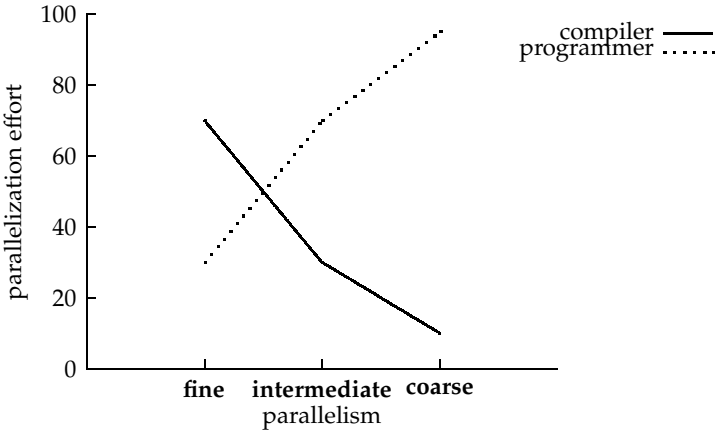


Figure 1: Depending on the level, parallelization and distribution strategies are supported differently.

According to the task the parallel program has to perform, three major models can be distinguished [Wal95]. Mixed forms are likely (→ figure 2).

Facts

- **data parallel**: The same algorithm can be applied to different parts of a partitioned data set simultaneously without dependencies. This usually is the case with programs for the *single program, multiple data (SPMD)* class, which is a relaxation of the SIMD class that is not restricted to executing the same processor instruction simultaneously, but allows more generally to run the same program on multiple data. Note however that asserting the no-dependency property frequently involves complicated duplication and synchronization of boundary information. Most distributed and parallel programs using message passing libraries fall in this category.

- **functional parallel**: Different steps of the same algorithm are applied to different parts of the data simultaneously. This usually is the case with pipelined algorithms. Note that it must be ensured to execute the steps of the algorithm in the correct sequence.

- **concurrent parallel**: Different algorithms operate on the same data to compute competing results. The first result is taken and the other algorithms still running are terminated.

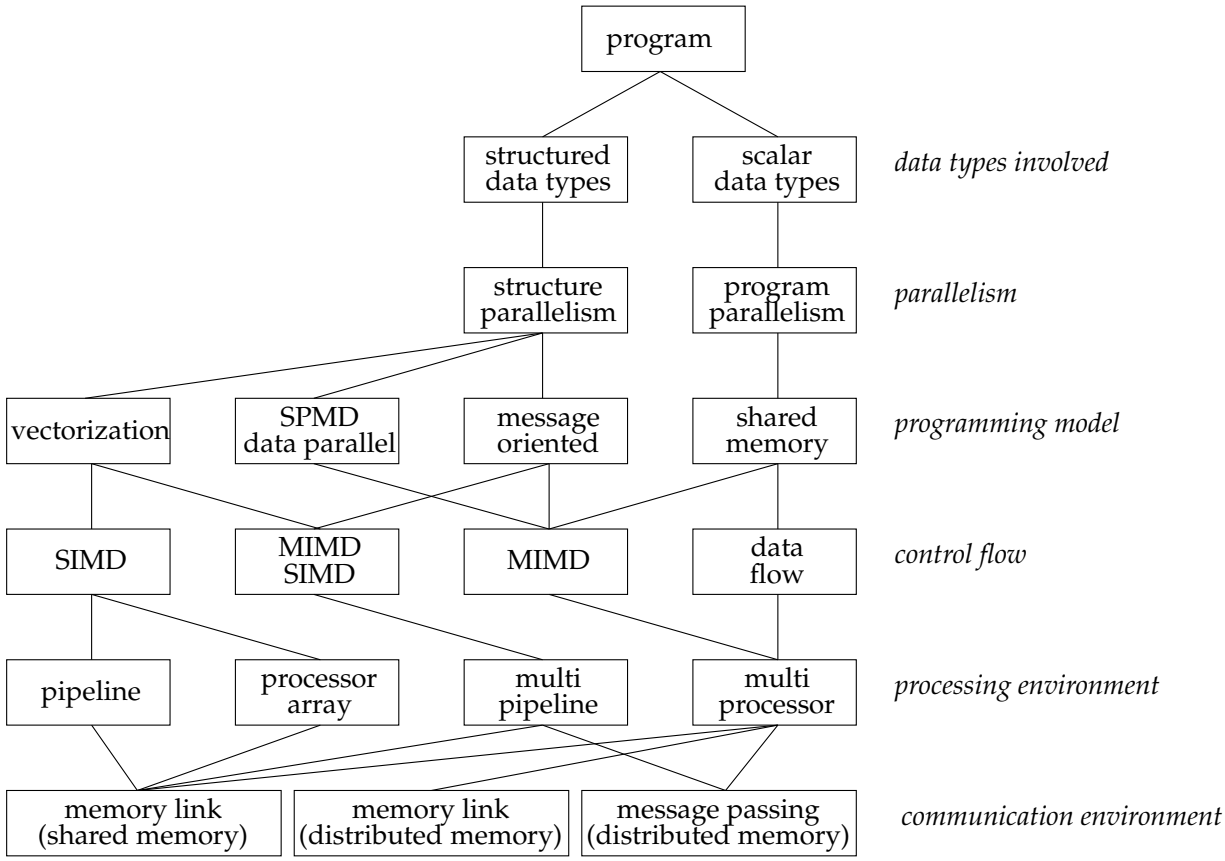


Figure 2: Classification of parallel computing architectures according to [Gil93].

Programming efficient parallel and distributed programs is not trivial. Additionally, Amdahl's law [Amd67] states that for every program there exists an upper bound of scalability because of serial program parts, limited communication bandwidth and process idle times for synchronization ($\rightarrow$ figure 3).

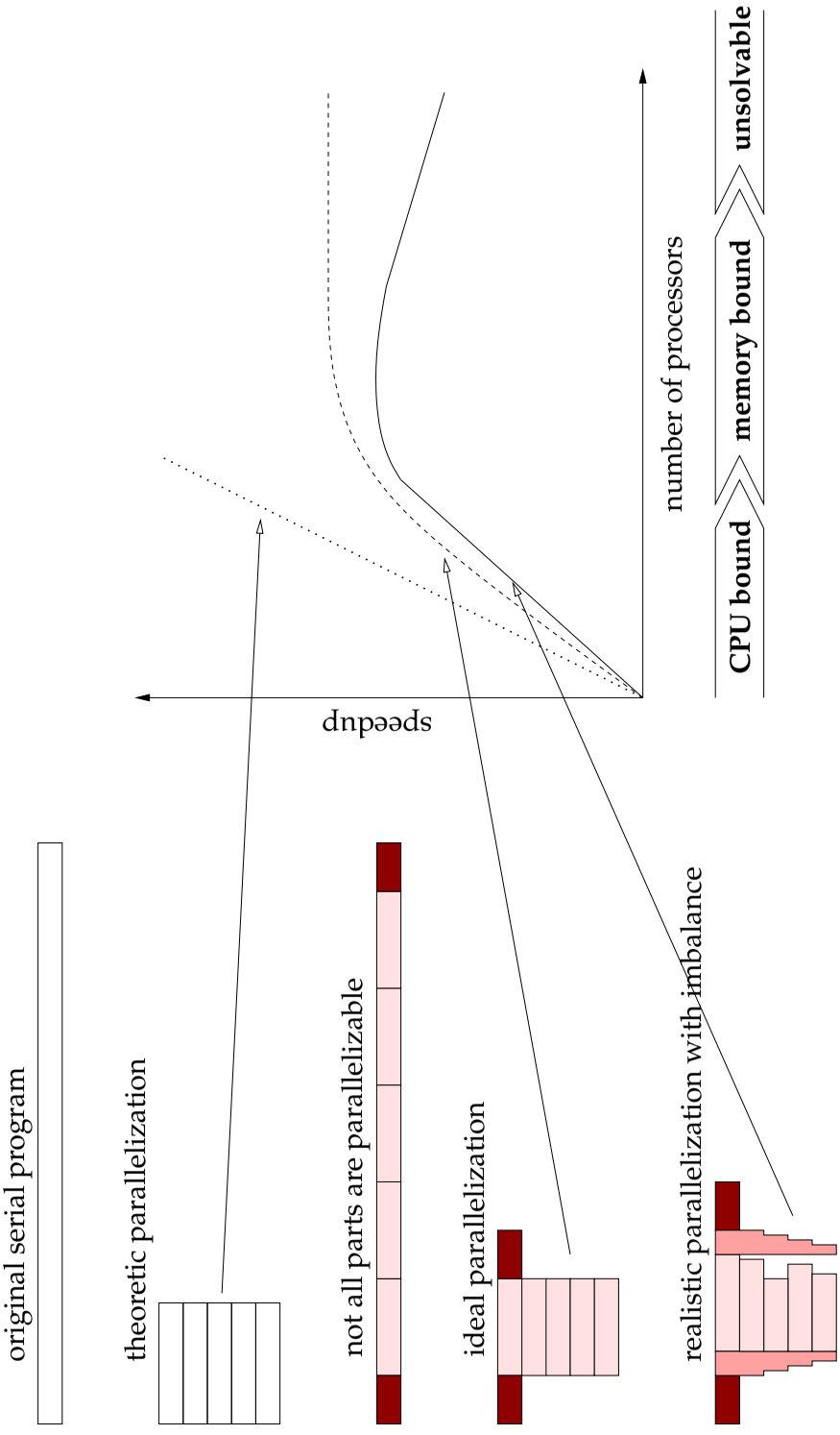


Figure 3: Amdahl's law states that scalability is not endless.

1

COARSE STRATEGIES

Form must follow function.

— Le Corbusier

1.1 Introduction

Whether one adds distribution and parallelization functionality to an existing system, or designs functionality that is to be integrated into a new system right from the start, there are some fundamental decisions to be made. In this chapter, we focus on concepts that implement these design issues.

All concepts provide a hierarchical layering of functionality in order to abstract from the actual hardware oriented implementation. They differ in the provided communication concept, the supported programming languages and the bridgeable distance.

Several object-oriented frameworks for supporting parallel or distributed programming have been suggested, e.g. POET [MA] or EPEE [Jez93]. POET is a C++ toolkit that separates the algorithms from the details of distributed computing. User code is written as callbacks that operate on data. The data is distributed transparently and user code is called on the particular nodes on which the data is available. Although POET as well as all other frameworks abstracts from the underlying message passing details, it requires to adapt the algorithms to the given structure of the framework and is thus not transparent to the programmer.

Other approaches view local resources only as a part of a possibly world-wide, distributed system (“computational grids”, “world-wide virtual computer”), for instance Globus [FK97] or Legion [GLFK98]. While these are certainly a vital contribution to distributed computing, the demands on the code are significant and by no means transparent to the programmer, which always should be the main goal of programming efforts. In the following, we present the most prominent message passing libraries and object-oriented communication frameworks.

1.2 Message passing systems

In order to communicate over address space boundaries, one of the oldest communication concepts in computer science is the *message passing* concept. The idea is to provide a system level facility that forwards blocks of data that have a specific sender and receiver. The communication must be started explicitly by the sender through providing the data block to transfer to a system call. For efficient implementation of the system call, the message passing is most often done in synchronous mode, which means that both sender and receiver have to issue a corresponding system call. Either the sender or the receiver is blocked until the counterpart reaches the synchronization point, at which the transfer occurs.

1.2.1 PVM

The *Parallel Virtual Machine (PVM)* [GBD+94] communication library offers the user a console for program control of a virtual machine consisting of a configurable number of real machines. The parallel programming model is the master–slave model where a designated master process is started that distributes the workload to the slaves and waits for their completion. The results are collected by the master for output. The creation of the slave processes therefore is equivalent to forking worker threads on shared memory machines. The PVM library takes care of distributing them to the participating hosts. The `pvm_spawn()` function creates the slave processes and returns identifiers for future reference.

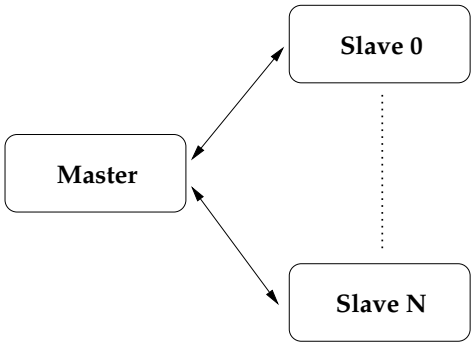


Figure 1.1: PVM offers the master–slave programming model.

Exchanging data between all participating processes is done by explicitly packing the data into a message buffer. PVM offers several calls to create message buffers and ensures a platform independent data format within the buffer. The message may contain data of different types, but everything must be packed into the buffer by PVM routines that usually perform a copy operation. The message can then be sent to another process by calling `pvm_send()` with the process identifier of the receiver. Recent PVM revisions also allow to attach a tag to the message for additional classification of the incoming message by the receiver, who has to unpack the data from the buffer. This again is a copy operation. Furthermore, the data layout in the buffer must be known implicitly to the receiver, as there is no query function or abstract declaration of the buffer content. The actual data transfer will only take place, if both sender and receiver call the appropriate PVM function. The processes are blocked until the peer process answers.

The usual program skeleton of PVM distributed applications therefore is rather rigid:

Master	**Slave**
① Create the slaves	① Wait for work
② Send work data to slaves	② Compute result
③ Collect results	③ Return result to master

1.2.2 MPI

The *Message Passing Library (MPI)* [MPI97] is a standardized definition of the message passing data transport paradigm. It has evolved since 1994 to the current version 2.0. It is supported by nearly every multi-processor environment.

MPI is most often used for implementing SPMD class programs. The number of participating processes is fixed and determined upon program startup by the library. As every process executes the same program code, MPI provides identification routines that allow each process to find its place in the processing environment and perform its task. The resulting typical pattern of an MPI program is shown in listing 1.1.

```
#include <mpi.h>

int
main(int argc, char *argv[])
{
  int npes,mype;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mype);

  if (mype == 0) {
    .... do some master thing ....
    MPI_Recv(...);
  }
  else {
    .... do some slave thing ....
    MPI_Send(...);
  }

  MPI_Finalize();
}
```

Listing 1.1: The typical MPI programming pattern.

Communication in MPI takes place between members of a *communicator*. Initially, all participating processes are sharing the `MPI_COMM_WORLD` communicator. In order to restrict communication to a group of processes, sub–communicators can be created. This is also very useful for library functions that may not interfere with the main programs message exchange.

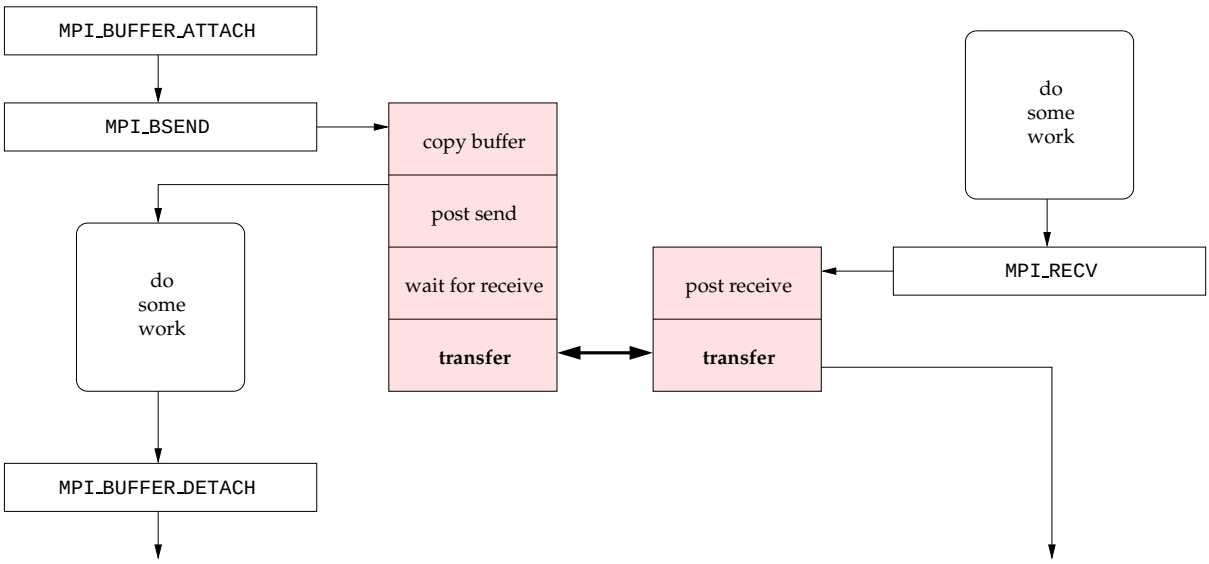


Figure 1.2: Control flow for blocking, buffered point-to-point message passing in MPI.

One goal of the MPI developers was to make the communication most efficient by minimizing the number of copying steps that take place and allowing to run computation and transmission in parallel. MPI therefore has three basic communication modes:

Facts

- **Blocking point-to-point** message transfer guarantees that the provided message buffer can safely be reused when the call completes. The MPI system can choose to copy the data to a system buffer (*buffered send* → figure 1.2) or directly transmit it to the receiving buffer (*synchronous send*). There are special MPI calls to enforce one of the methods.
- **Non-Blocking point-to-point** message transfer calls return immediately. The provided data buffer may not be used by the program until MPI has completed the transfer. This condition can be queried selectively for specific messages. The transfer again can be either buffered or synchronous and a specific method can be enforced by appropriate calls.
- **Collective communication** operations for broadcasting, gathering and scattering of data and simple barrier synchronization. The participating processes are determined by the current communicator scope.

Note that MPI distinguishes the handling of the message buffer (blocking ↔ non-blocking) from the actual data transfer operation (buffered ↔ synchronous). It can therefore satisfy several different communication paradigms of an application. However, the fundamental requirements for a client, which are a fixed number of processes and knowing the receiver of a message, cannot be circumvented. MPI is also very limited in its interoperability.

By far the most common field of application for MPI are simulation codes for supercomputers. Although these architectures most often do not supply fast shared memory communication, it is common that there are MPI implementations tuned to a specific machine by the manufacturer in order to exploit fast node interconnects. As MPI is one of the few well established standard libraries on supercomputers, the consequence is that writing somewhat portable supercomputer applications is equivalent to employing MPI.

G page 145

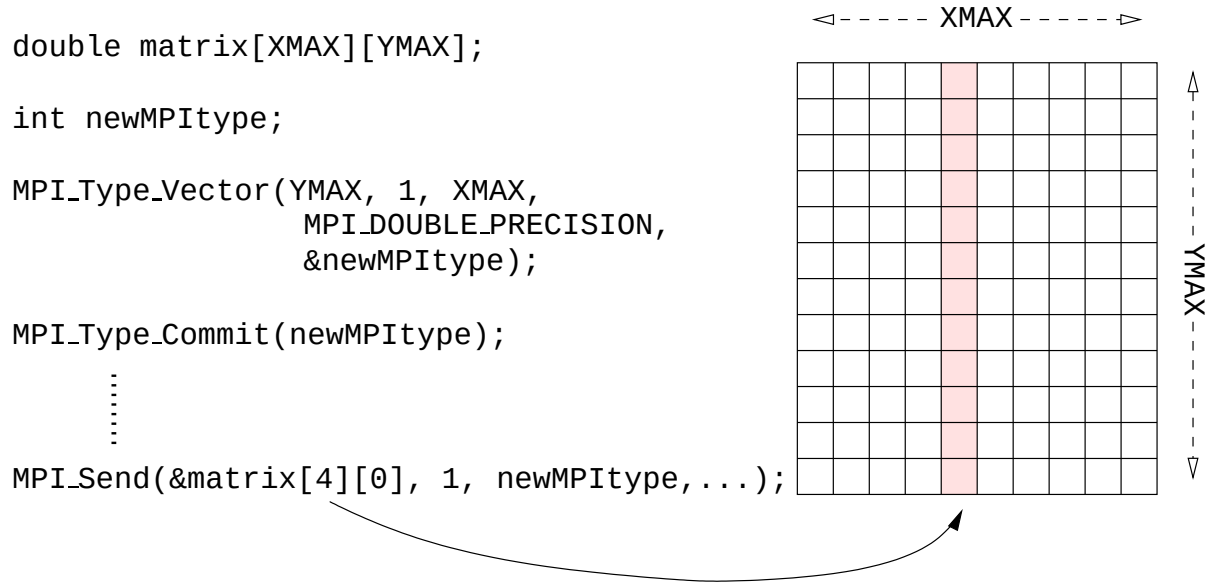


Figure 1.3: Deriving and using new data types for MPI message passing.

A MPI message must consist of one or multiple instances of a single data type. MPI declares symbols for all primitive data types. In order to be able to send mixed data type messages, one needs to define a new *derived MPI data type* that is built from known primitive data types. There are several construction functions that declare and install the new data type at runtime. Once defined, it can be used like any primitive type, and MPI will take care of representation compatibility of floating point parts and endianess of integer parts. A type declaration essentially works like declaring a C struct that fits the data layout. Additionally, it can contain strides for padding. Figure 1.3 shows an example of how to declare a new vector type that represents a matrix column. Using the new type, a column vector can be transferred by simply specifying the top column element and the new type identifier to the MPI calls.

For advanced communication patterns, MPI has additional functionality:

Facts

- **Message probing** for checking whether a specific message is receivable.
- **Persistent communication** for optimized repeated communication (for the lifetime of the program).
- **Thread management** at runtime.
- **One-sided communication** for simple asynchronous update of specific memory locations.
- **Parallel file I/O** on parallel (block striped) file systems to circumvent yet another supercomputer bottleneck.
- **External interfaces** to other languages and communication libraries.
- **"In place" buffer** usage for employing the same buffer for send and receive operations.

Unfortunately, most of the interesting features on the above list are not supported with current MPI implementations. They are part of the MPI 2.0 standard which is not yet fully implemented on current supercomputers.

1.3 Distributed object-oriented systems

The basic idea of object-oriented programming to encapsulate data and algorithms carries over nicely to distributed systems. The "naturally" separated address spaces of a distributed algorithm strongly suggest object-oriented design, which in turn enforces well defined interfaces. Because data transfer between address spaces or networked computers is not trivial, it must be handled in a consistent and extendable way for all participating objects to make it comprehensible for the programmers. Hierarchical structures must be available to foster the reuse of existing functionality, minimize the development effort for implementing a higher level interface and to abstract from implementation details, like available hardware.

1.3.1 ACE

The task of the *ADAPTIVE Communication Environment (ACE)* [Sch94] is to provide a unified interface for basic communication mechanisms on different platforms. The abstraction comprises simple encapsulation of available system calls and software solutions for calls not available on the current platform.

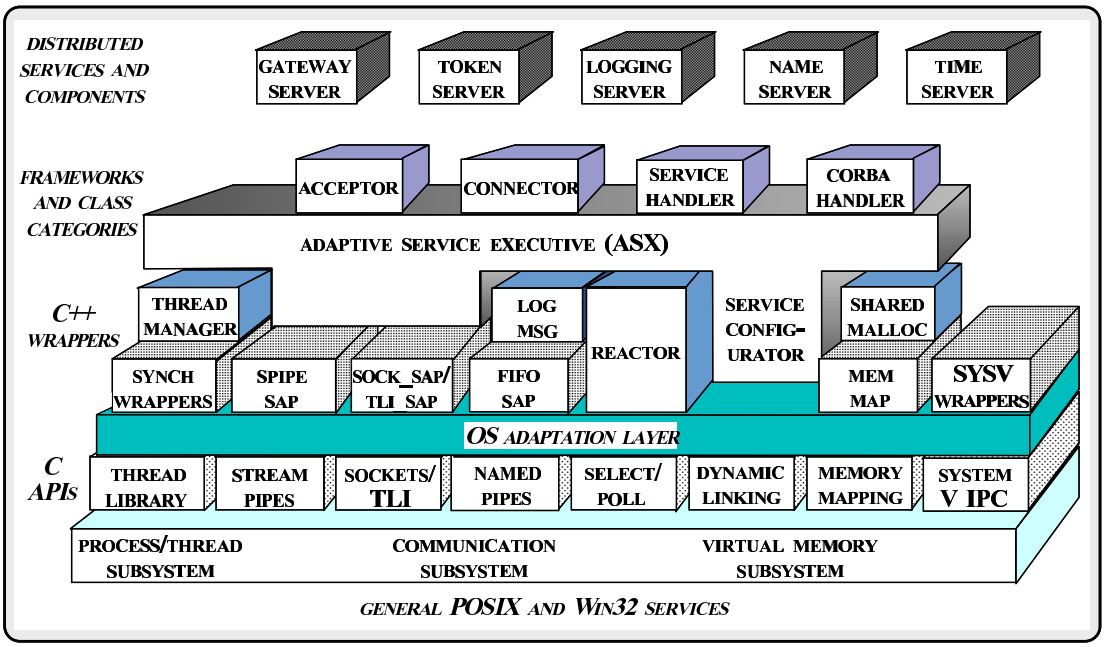


Figure 1.4: The hierarchical architecture of the ACE framework. Figure taken from [Sch94].

All functionality is implemented with strongly typed C++ interfaces. Therefore, the ACE framework allows much better syntax checking and more consistent C++ programs than using the C system calls directly. The system calls are bundled within the *OS Adaption Layer*. As figure 1.4 shows, it provides platform independent support for thread control, synchronization mechanisms, inter-process communication, event demultiplexing, explicit dynamic binding, memory mapped files and management functionality for shared memory areas. Implementations of the OS Adaption Layer exist for many UNIX dialects, the Win32 API, OpenEdition MVS and several real–time operating systems (VxWorks, LynxOS,...).

On top of the OS Adaption Layer, there are *C++ Wrapper* for higher order routines for communication, synchronization and virtual memory handling. They are of great value for writing reusable code, developing reentrant objects and avoiding common thread synchronization pitfalls.

G page 145

ACE has been carefully designed for generating highly efficient code. There is almost no performance penalty compared to the direct usage of the system calls. The achieved type safety is therefore of even greater value. The ACE framework uses for the implementation of the C++ Wrappers many of the *design patterns* described in [GHJV95].

Here is a short overview of the ACE services:

Facts

- Generic atomic operations (barriers, recursive mutexes, tokens, timers)

- Automatic semaphores (locks, guards, conditions)
- Thread Manager / Thread specific storage
- Event de-multiplexing
- Service initialization and configuration
- Layered service streams
- Middleware applications (CORBA ORB adapter)

1.3.2 CORBA

The *Common Object Request Broker Architecture (CORBA)* [OMG97b] is at the heart of the *Object Management Architecture (OMA)* [Sol95], an open, object-oriented middleware architecture defined by the *Object Management Group (OMG)* [OMG97a]. The goal is to enable the collaboration of distributed software components without hardware, operating system and programming language boundaries. Only the interfaces and the communication semantic is defined by the OMG. There is no prescribed implementation. On top of the CORBA definition, *services* and *facilities* [ACG97] are defined.

G page 145

There exists a large variety of CORBA implementations for several object-oriented languages. In the following, we discuss the CORBA concept by looking at TAO [TAO97], which is a freely available ORB implementation using the ACE framework library [Sch94].

The OMA defines two basic models [OMG96]:

① **Object model:** It describes distributed objects independent of a programming language, although the representation has a strong C++ flavor. An object is defined to be a closed, uniquely and invariably named entity. The provided services can only be accessed by uniquely defined interfaces. A *client* can send a *request* to an object to have the advertised service performed for him. The implementation and location of the *server* is not known to the client. The role of client and server is defined by the execution of the request. A server can act as a client to other objects.

② **Reference model:** It describes the way in which the participating objects communicate with each other and with the services and facilities of the system.

Object Request Broker (ORB): The ORB is the central functional entity of CORBA. It provides the communication mechanism for the participating objects and controls their life-cycle and configuration. In colloquial language it is therefore often identified with CORBA. The components of an ORB are (→ figure 1.5):

- *ORB Core*: This is the central transport channel between client and server objects. As all clients connect to it, the location, the implementation, the state and the communication routines of any server are transparent to a client.

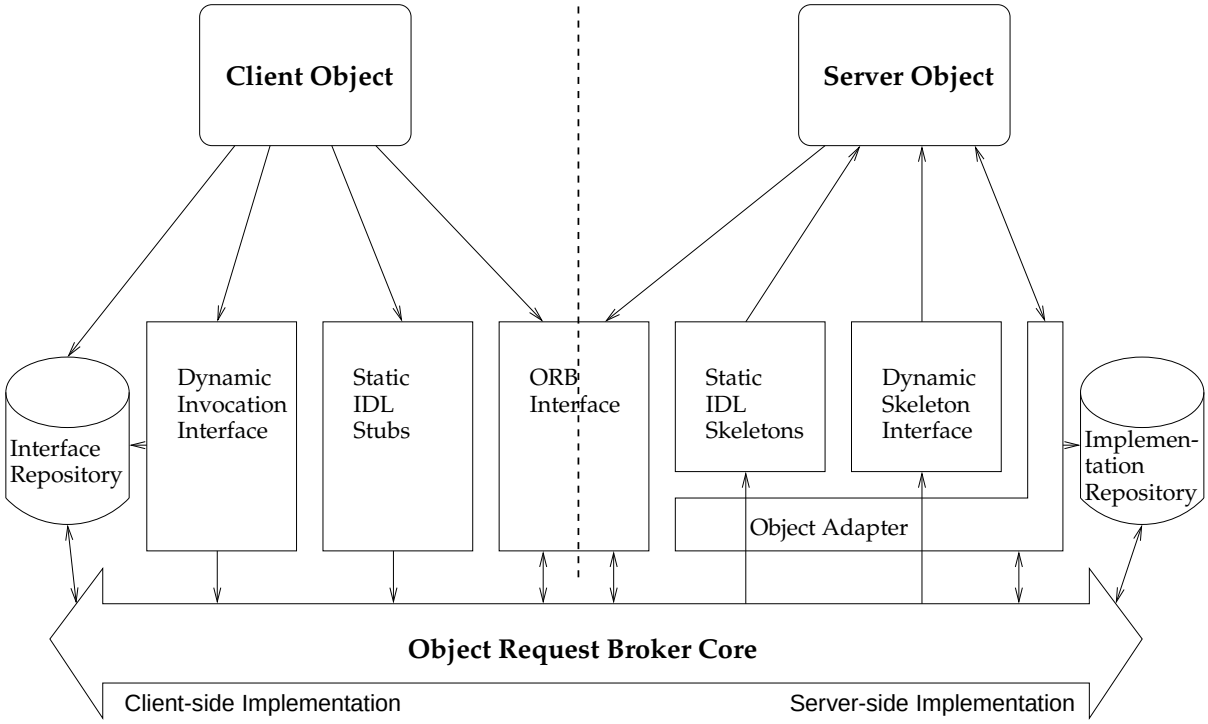


Figure 1.5: Architecture of the ORB according to the CORBA specification version 2.0.

- *Interface Definition Language (IDL)*: It is a purely declarative language for describing the interfaces of the participating objects. It is independent of programming languages and their data structures.
- *Language Mapping*: Predefined rules for mapping the IDL definitions to interfaces of higher level programming languages. The mapping is performed by the IDL compiler.
- *Stubs and Skeletons*: They are created in some specific programming language by the IDL compiler. Client–side stubs implement the creation and sending of requests, while server–side skeletons are for receiving the data and performing a method call on the server object. Together, they implement a synchronous server invocation with transfer of parameters and result data (*marshaling*). Server methods, that do not throw *exceptions* and do not return a result, can be declared to be `oneway` in the IDL language. Software manufacturers of an ORB are free to allow asynchronous communication for these requests.
- *Interface Repository*: A database for runtime type information for the server interfaces created by the IDL compiler.
- *Dynamic Invocation and Dynamic Skeleton Interface (DII / DSI)*: When ex-

ecuting requests, whose signatures are created by the client at runtime (*dynamic dispatch*), the static stubs and skeleton interfaces can not be used. The DII and DSI interface allow the execution of such requests.

- *Object Adapter*: It is the connection between a server implementation and the ORB. It registers the server object with the ORB upon startup, creates references to the method implementations, instantiates objects upon request, identifies routed requests (*de-multiplexing*) and executes the method called by the request.

  The OMG has defined two basic object adapter types: the *Basic Object Adapter (BOA)* and the *Portable Object Adapter (POA)*. The POA has much more functionality and is discussed in detail in section 1.3.4.

- *Inter–ORB Protocol*: The *General Inter–ORB Protocol (GIOP)* defines the interface for collaboration of ORBs. The *Interoperable Object References (IOR)* format defines standardized messages for communication of ORBs of different manufacturers over abstract channels. For example, the *Internet Inter–ORB Protocol (IIOP)* implements using the GIOP over a TCP/IP network.

Object Services: The OMG has defined in the *Common Object Services Specification (COSS)* [OMG95b] several system-related, horizontally oriented, universal services. They are realized as a module with an IDL interface and are therefore seamlessly integrated. The most important ones are Life Cycle Service, Persistence Service, Naming Service, Event Service, Transaction Service and Trader Service.

G
page 145

Common Facilities: Like the object services, the facilities are defined as modules with IDL interfaces [OMG95a]. Unlike them, they do not exist on the system level, but are directly used by the programmer. The most important facility is the System Management Facility [OMG95c], that offers configuration, runtime management and monitoring.

Domain Interfaces: They declare components and interfaces that are optimized for specialized application areas (medicine, telecommunication,...).

Application Interfaces: They are the interfaces of the new application, formulated as IDL interfaces. Of course, they are not standardized by the OMG.

1.3.3 TAO

The ACE ORB (TAO) [TAO97] is an implementation of a CORBA ORB using the ACE framework. TAO extends the CORBA specification with domain interfaces for *real–time applications*. In the area of telecommunication, system control and WWW, applications are inherently distributed. Using existing middleware, like standard CORBA, however fails to deliver the vital *Quality–of–Service (QoS)* paradigm. TAO offers an integrated solution by providing

- specification of a QoS
- enforcing and controlling the QoS
- support for creation and application of real–time functionality
- allowing non–portable optimizations for special cases

Figure 1.6 shows on the left side the general approach of the TAO ORB when building a connection: Beside the actual data link for the CORBA services, the client and the server negotiate the QoS definitions.

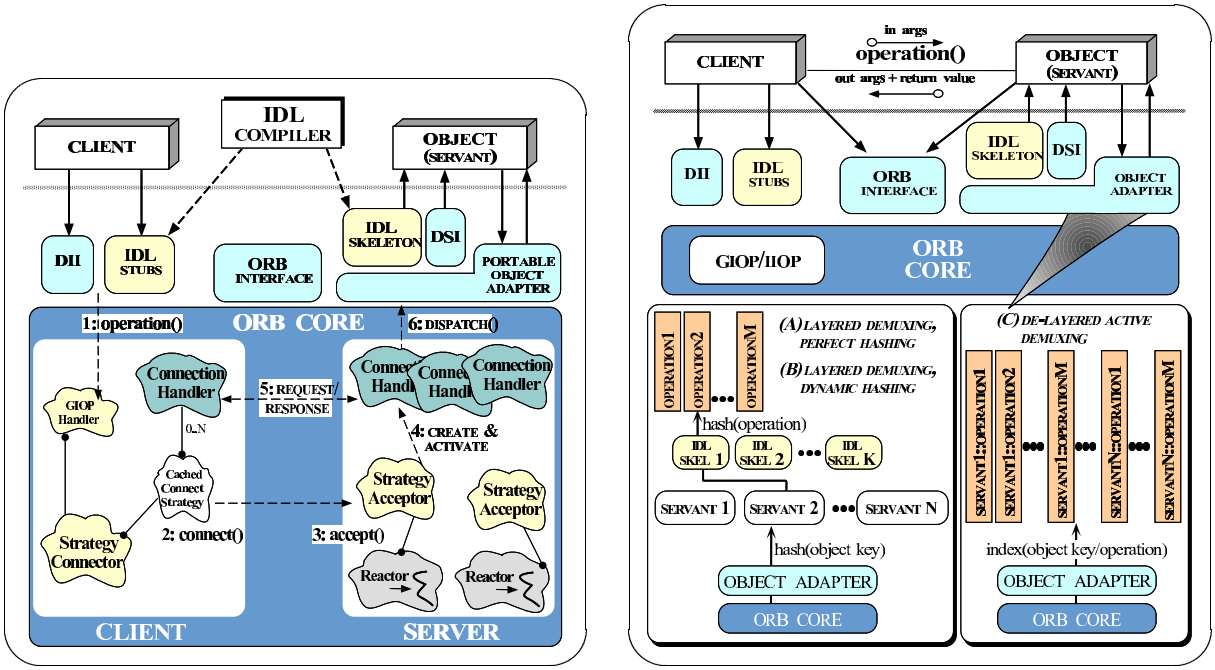


Figure 1.6: The TAO architecture — current status. Figure taken from [TAO98]

TAO handles the definition of attributes of a connection within a *Connection Strategy*. It comprises

- **Transfer of execution priority** between the implementations of client and server (end–to–end). The priority classes of several computer operating systems are mapped into a portable scheme. Clients can bequest their priority, servers can prescribe the usage of a priority class for access.
- **Connection properties**, for example the TCP/IP protocol, can be selected and configured. The GIOP can be configured using preferences.

Facts

- **Management of parallel routines** (*thread–pool*) is supported. Threads can be pre-allocated and configured (stack size, priority, static data, maximum number of threads), on the ORB level and on the object adapter level (→ section 1.3.4).

- **Explicit binding** of implementations by early establishment of priority–driven or dedicated connections is possible according to the capabilities of the selected communication protocol.

- **Protection against reentrant execution** (*mutex*) in a portable way to eliminate race conditions of synchronization mechanisms in the ORB and the actual application.

1.3.4 The POA object adapter

The *Object Adapter* is the main part of the server side of CORBA. It creates references for all of the registered objects, activates and removes implementations of the objects, assigns requests (de-multiplexing) and creates the connection between the IDL skeletons and the ORB.

The OMG has specified two adapter types by now: the *Basic Object Adapter (BOA)* and the *Portable Object Adapter (POA)*. The POA allows to create server implementations that are portable between different ORB implementations, which was the major drawback of its predecessor, the BOA. Because the OMG has also redefined some other details, the two adapters are incompatible. The BOA has therefore been dropped in the CORBA specification version 2.1.

The most important design aspects of the POA are

Facts

- **Portability:** The POA enables the usage of server implementations with ORBs of different manufacturers.

- **Persistence:** The POA supports persistent objects by ensuring that the object gets the same identity if it advertises a service that exceeds its lifetime.

- **Automation:** Objects are invoked transparently, i.e. the implicit usage of an object reference triggers the activation of the object.

- **Saving resources:** One can assign multiple identifiers to a CORBA object. This can result in significant memory saving, if for example a database is modeled by declaring every record as a CORBA object.

- **Flexibility:** The POA can be configured freely. The identity of objects, the object state, the request processing (event–handling) and the existence of objects can be handled automatically by the POA or explicitly by the server class.

- **Behavior control (policy):** The handling of objects and requests (multi–threading, retention, lifespan, ...) can be specified upon creation of the POA.

- **Hierarchy:** One server can host multiple POAs as an adapter hierarchy. Each adapter provides a name space for contained objects and adapters. It implements recursive functions, for example deep copy, recursive delete ...
- **SSI / DSI Integration:** Server implementations can be derived both from static skeletons and DSI, transparently for the client. Two clients with identical interface therefore can choose freely which server interface to use. This also applies in temporal context: A client can use the static interface first and later switch to accessing the server by the DSI.

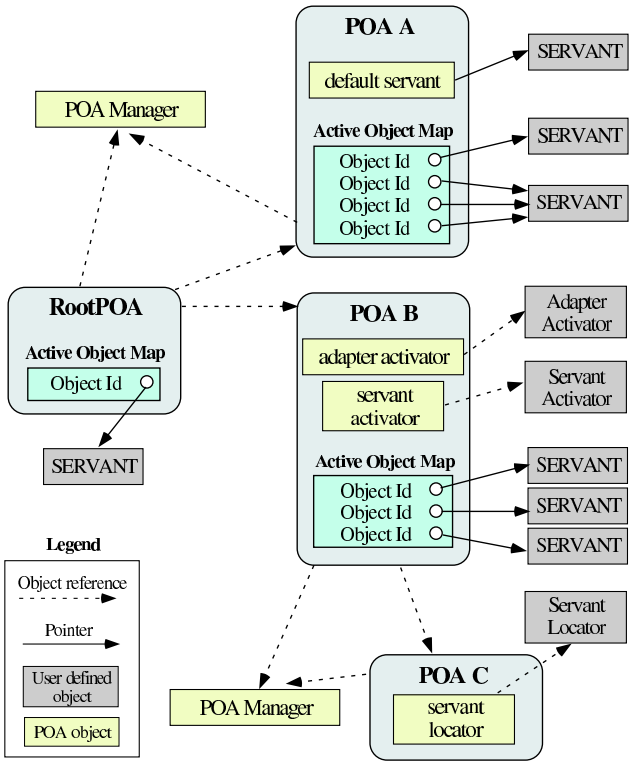


Figure 1.7: Using the Portable Object Adapter. Figure taken from [PS98].

Figure 1.7 shows the architecture and the typical usage of the POA. The abstraction the POA builds is visible only to the server. Client requests must be routed correctly to the server implementation. On the right side of figure 1.6, the de-multiplexing techniques of the BOA (A) and (B) are contrasted to the *active de-multiplexing* (C) of the POA. It enables a more efficient mapping of server functionality for real–time purposes. The encapsulation of different parts of the server implementation is reflected by the adapter hierarchy. The *RootPOA* exists in every ORB and can be queried by the server implementation. Object identifiers with the selected retention–policy are stored in the *Active Object Table* of the derived POAs. The *POA Manager* controls the life-cycle of the POAs.

1.4 Discussion

Both the message passing concept and the object oriented concepts have different strengths and weaknesses. The following table 1.1 does not try to argue in favor or against one of the concepts, but rather presents a collection of typical questions that must be taken into account when planning parallel and distributed applications or libraries.

The bottom line of this chapter is, that choosing the coarse distribution strategy has a large impact on the overall library and application design. The pros and cons therefore have to be evaluated very carefully keeping the primary constraints the application has to meet in mind. Because of its normative character, coarse distribution and parallelization strategies get deeply embedded in the application logic, which means that they cannot be removed from or introduced to existing code easily. The coarse distribution and parallelization strategy that is used therefore must be among the first considerations a developer of a library or a program has to decide upon.

As nowadays large computation facilities, be it supercomputers or PC clusters, offer several levels of parallelization, the coarse distribution strategies have to be complemented with finer grained concepts to efficiently use the available resources. The coarse concepts are therefore often applied to the topmost hardware architecture layer, like node interconnects or high-speed LANs. For the application developer, it is therefore important, that the coarse distribution strategy chosen does not interfere with parallelization concepts on a finer level.

In the following chapter, we present strategies for efficient exploitation of parallel resources. They can be perfectly embedded with the coarse strategies presented in this chapter.

	PVM	MPI	CORBA
Programming model	One master process, any number of slave processes.	Fixed number of processes.	Not process based.
Library binding	Compile time	Compile time	Runtime
Data volume per message	Max. transfer buffer size	Max. data struct size	Streaming possible
Programming language	All (nearly)	All (nearly)	Object-oriented only
Asynchronous transfer	No	Yes	Yes
Data types	Pack/Unpack calls for only a few basic types. Mixing of types in buffer possible.	Calls for all basic types. Self defined struct types possible. Buffer must have a single type.	No type restriction, including recursive structs and arrays.
Optimized transport	No	Shortcuts for special cases, broadcast and integrated computation functions.	Hints for transport direction.
Persistent state	No	No	Several facilities and services.
Dynamic server binding	No. Single program.	Yes, within compiling single project, but not interoperable.	Yes, also across projects, only interface declarations needed.
Communication scope	Always full domain.	Configurable subdomains (communicators).	Does not apply. Server location not known to clients.
Bridgeable distance	Local cluster/machine.	Local cluster/machine.	Any distance. Gateway server for inter-LAN communication available.
Development effort	Programming of message transfer. Writing of explicit code. Linking of library.	Exact programming of every message transfer. Writing of explicit code. Linking of library.	Writing of interface declaration. Code generation by IDL compiler. Linking of library.
Mixed programming language	No	Yes	Yes
Process migration	Possible by explicit intervention of master process.	No	Yes, also external steering through ORB service.

Table 1.1: This table compares several aspects of coarse distribution and parallelization concepts that must be considered when planning libraries or programs.

2

INTERMEDIATE STRATEGIES

Programming is understanding.

— Kirsten Nygaard

2.1 Introduction

In the last chapter, several coarse distribution and parallelization strategies have been presented. They all have in common, that they handle the communication between threads or processes explicitly. This promotes a good understanding of when and how message transfer occurs during program lifetime. Because of their dedication to bridging large address space distances, the preferred application scope are coarse distribution tasks on the top architectural level. This makes them inefficient for more closely coupled communication tasks, like they occur regularly in shared memory environments. Therefore, they must be complemented with intermediate distribution and parallelization strategies.

As already pointed out in the introduction to this part of the thesis, there exist a number of large scale multi-processor machines that support the shared memory programming paradigm, although the memory is not local to all processors. With considerable hardware expenditure, this is achieved transparently for the programmer on ccNUMA machines. On NORMA architectures, (system) software can make the memory appear to the programmer as one shared address space. They are called *virtual NUMA* systems, or *distributed shared memory (DSM)* systems. Memory access is handled transparently by the DSM software and therefore allows to use the simple NUMA programming model on less expensive NORMA architecture hardware.

Initial DSM approaches were based on copying the memory page accessed by the program to local memory. Although this is a very cheap operation, it suffers tremendously from the *false sharing effect*: If two data objects are located on the same memory page, but are accessed by different processors, the page is permanently relocated. This

effect is getting more severe, as the memory page size tends to grow rapidly in modern systems to enable nowadays huge main memory sizes. Additionally, most operating systems do not allow the user to prescribe a specific memory page for data storage, so the effect cannot be controlled by the programmer. Modern DSM approaches allow to control the memory sharing on a data–oriented basis, where the data entities may be smaller than the memory page size. They do not build on the memory management hardware, but introduce memory access routines transparently and directly into the program by using a specialized compiler. The most prominent DSM systems are the virtual Orca machine [BK93] on the CM-5 architecture and the Shasta system [SGT96] for DEC Alpha clusters.

Altogether, intermediate distribution and parallelization strategies always try to provide a simpler view of the whole system for the programmer. The shared memory programming and communication model and a virtual multi-processor environment enable easy functional–parallel or data–parallel programming. In the following sections, we therefore present two strategies that have proven to be useful for many tasks.

2.2 Pipelining

Virtually every algorithm is composed of distinct processing steps that have to be carried out in sequence. Most often, the steps only communicate via intermediate result data. If an algorithm can be decomposed to satisfy this criterion and has to be applied to a large number of input data, *pipelining* it should be considered.

A pipeline is a concatenation of functionally independent processing modules. It is therefore also referred to as *functional decomposition* of an algorithm. Each module takes a certain data format as input and produces some result. In order to make the pipeline work, the output format of a module must match the input format of the following processing step. The crucial difference between a modularized program and a pipeline is, that each step of the pipeline can be implemented and run separately, i.e. one pipeline step does not use resources of another one.

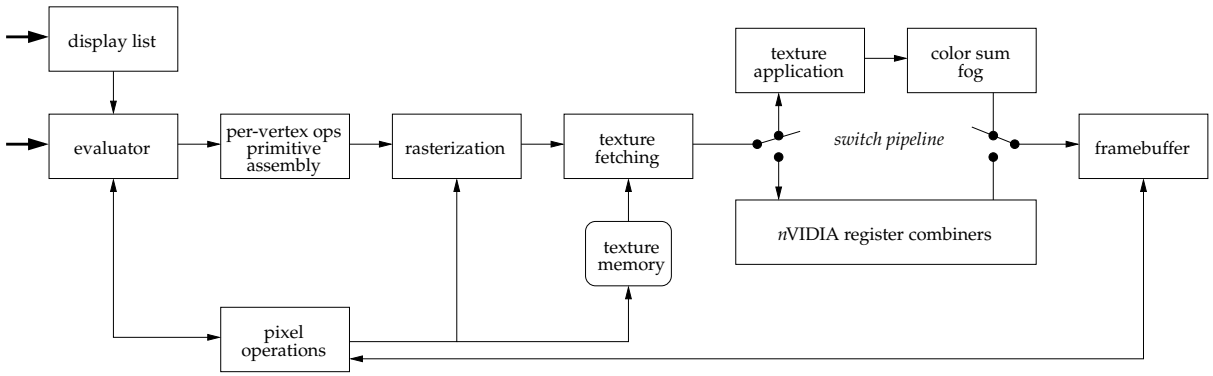


Figure 2.1: The *n*VIDIA OpenGL processing pipeline.

The usual way to drive a pipeline is to run it in synchronous mode. That is, all pipeline steps exchange data at the same rate. The pipeline therefore produces output data in every pipeline clock cycle. As this property is very attractive for hardware design, the pipeline concept is the classic approach for composing complex circuits. As the input is fed into the pipeline at the same rate as the internal data exchange, the first N cycles produce illegal output data, where N is the number of pipeline steps. In order to flush the pipeline, N dummy pieces of data have to be appended to the actual input data. Figure 2.1 shows the OpenGL processing pipeline on *n*VIDIA graphics boards as an example: For the texturing and shading step, there are two implementations that can be switched by the user. For the input data (the scene graph) and the output data (the pixels), this is transparent.

The pipeline concept is also often used in pure software implementations. Here, the pipeline steps need not to be run synchronously, as exchanging the intermediate data is done by transmission concepts that enforce a synchronization, like shared memory areas, callbacks or simple shell pipes. Decomposing an algorithm into pipeline steps can be nevertheless sensible, because of the abstraction and modularization introduced. The biggest benefit however is the possibility for parallelization and distribution. Because the pipeline steps have well defined interfaces and have private resource management, functional-parallel distribution is very easy. The most important point however remains valid also for software pipelines: The pipeline concept only makes sense for large amounts of input data of the same type.

One crucial point of creating pipelined processing in software is, to ensure that the pipeline steps are really executing independently. Otherwise most of the benefits will be void. If the creation of independent processes is not viable, multi-threading can be a good alternative. In any case, direct access routines that make assumptions about internal state of another pipeline step must be avoided. Although extending a more or less modularized program to a pipeline processing environment can be tedious, the advantages are significant. In table 2.1 we compare modular program design with the pipelining strategy.

2.3 OpenMP multi-threading

The most effective strategy to achieve a good speedup for data-parallel tasks on shared memory systems is multi-threading. The *OpenMP* [OMP02] standard supports this strategy by defining several functions for running and controlling parallel threads and for dividing up the work. Although there also exist OpenMP implementations that can work in a distributed memory context, the original idea was to enable highly efficient multi-threading. OpenMP also has library functions that let the participating threads identify themselves.

G page 145

OpenMP is a directive-based parallelization strategy. The directives are stated as compiler pragmas. This means, that OpenMP needs to be supported by the compiler.

Fortunately, the OpenMP standard is already well established and many compilers support it for a wide range of platforms. The OpenMP directives structure the program into parts that are executed in a certain fashion. As this procedure does not introduce new code fragments, the program can still be compiled without change using compilers not capable of OpenMP: They will simply ignore the unknown pragmas. The standard makes sure that this is true for all OpenMP pragmas. Library functions can be excluded from compilation by using the preprocessor macro `_OPENMP` that is declared in the OpenMP header file.

Each OpenMP-enabled program starts off single-threaded like any serial program. This continues to be so until a program part is reached that is marked as parallel. Using the C/C++ programming language, the part is simply identified by writing the corresponding OpenMP pragma in front of a structured block for which it should be valid. Upon entering a parallel part, OpenMP executes the code simultaneously on all participating threads. The number of threads is defined by an environment variable. When leaving the parallel part, the program returns to serial execution. An implicit synchronization is performed at this point, so the serial execution starts after all threads have completed.

```
double pi = 0.0;
double w = 1.0 / n;

#pragma omp parallel private(x,sum)
{
  double sum = 0.0;

#pragma omp for
  for (int i=1; i<n; ++i) {
    double x = w * (i - 0.5);
    sum = sum + f(x);
  }

#pragma omp critical
  pi = pi + w * sum;
}
```

Listing 2.1: A multi-threaded program that uses OpenMP directives.

Within a parallel part, program variables can be private to a thread or can be shared between all threads. The rule here is simple: All variables in the static extent of the parallel part are shared, variables in dynamic extent and variables of subroutine calls are private to the thread creating them. The static extent is simply the lexically enclosed region of the parallel part. For accessing shared variables by multiple threads, synchronization mechanisms are available.

Listing 2.1 shows a small OpenMP-enabled program. Note that the compiler pragmas do not change the program logic, so translation using a non-OpenMP compiler is possible without changes. The program first declared two variables. As this is done before entering the `omp parallel` part, the variables will be shared. Inside the parallel part, the statement `omp private` declares the variables `x` and `sum` to be thread-private. They act as temporary variables for each thread that can be accessed without need for synchronization. The `#pragma omp for` finally divides up the work the loop has to do in a way, that each thread gets assigned some of the iterations to perform. There are several scheduling strategies that can be used. The default is to do a static partitioning, so thread $t = 1 \ldots N$ will perform iterations $0 \leq i < \frac{t*n}{N}$, thread $t+1$ will perform iterations $\frac{(t-1)*n}{N} \leq i < \frac{t*n}{N}$, and so on, with N being the number of participating threads. Finally, the result is accumulated in the variable `pi`. As this variable is shared among all threads, the `#pragma omp critical` ensures, that only one thread enters this region at a time.

OpenMP has a number of additional features that help the programmer with common parallelization problems. Here is a short overview:

Facts

- **work distribution** for loops with fixed number of iterations (`for` loops with computable terminator) and for independent serial sections of a program.
- **reduction** of thread-private data to a shared variable with implicit synchronization. The reduction operation can be specified for each variable separately (+, *, AND, OR, MAX, MIN, ...).
- **scheduling** for work distribution can be static, dynamic (round-robin) or deferred until runtime.
- **explicit synchronization** as barrier function or single/master blocking. The latter will halt all threads except a designated one that executes the marked part. When the part is complete, all threads resume.
- **library routines** that allow to retrieve the number of participating threads, the number of processors available, the thread's own identifier, whether code is currently executed in parallel and adjusting the number of participating threads dynamically.

2.4 Discussion

Intermediate distribution and parallelization strategies complement the coarse strategies with concepts for efficient work sharing. This strongly suggests using them especially for the shared memory model. As modern supercomputer architectures offer

distributed memory on multi-processor nodes, the intermediate strategies are most useful for fast intra-node communication. DSM systems and distributed memory OpenMP versions allow to use the code seamlessly on several other architectures.

In order to help the programmer to make up his mind when designing a parallel program, table 2.1 compares modularized program design to the pipeline concept. As with the coarse strategies discussed in section 1.4, the counter-value of using the one or the other concept varies considerably with the concrete application.

PRO / CON

	program modules	**pipeline steps**
Parallel execution	Dangerous. Methods and functions must be reentrant or have to be synchronized explicitly by separate locking mechanisms.	Simple. Pipeline steps do not share resources.
Data exchange	Simple. The successor module reuses data structures built by its predecessor.	More elaborate. If implementations of the pipeline steps are not developed together, the data format of the intermediate results must be available at compile time and plausibility checking must be performed by each step.
Efficiency	Very good. No overhead.	Good. Bandwidth of data transport channels must match data production rate of pipeline steps.
Flexibility	Good. Modularization promotes code reuse, but enforces adherence to internal structures.	Very good. Data exchange format is often very simple. Steps can even be exchanged at runtime. Internals of one step are transparent (pure software or hardware accelerated).
Scalability	Poor. Parallelization and distribution has to be programmed explicitly.	Good. As the steps are independent, parallelization through replication is easy. Flow control steps ("multiplexer") are simple to program and can be generic.
Distribution	Not difficult, but requires intervention into program logic. Functionality for data transport is mixed with computation functionality.	Easy. Data transport is kept external to computation steps. Standard stream based data transport can be used. Transport system can be exchanged without notice to pipeline steps.
Graceful degradation	No. Failure leads to about of entire system.	Yes. Pipeline can be supervised externally and dead steps can be restarted without influence to others.

Table 2.1: This table compares several aspects of modular program design and pipelining that must be considered with respect to distribution and parallelization.

While using the pipeline strategy for intermediate distribution and parallelization is

surely questionable for some application cases, using OpenMP is nearly always superior to POSIX multi-threading. The following list summarizes the advantages of OpenMP:

PRO CON

- ✓ The compiler can optimize the handling of thread-private memory.
- ✓ There is a tool set for correctness checking, automatic parallelization and tuning.
- ✓ Parallelization can be incremental without code modifications.
- ✓ There are OpenMP implementations for distributed memory systems.
- ✓ Integrated work sharing concepts for loops and independent serial code blocks with several scheduling options (“don’t invent the wheel anew”).
- ✓ Integrated platform independent synchronization functionality.
- ✓ Parallel variable reduction operations.
- ✓ “Invisible” in source code written for coarse strategies.
- ✗ Needs OpenMP-capable compiler.

3

FINE STRATEGIES

Premature optimization
is the root of all evil.

— Donald E. Knuth

3.1 Introduction

After coarse and intermediate parallelization and distribution strategies have been applied successfully, one should strive for optimal numerical performance. For many implementations, the observation is that the average rate of *floating point operations per second (FLOPS)* that are executed by a program is in most cases below 50% of the theoretical peak performance of the processor. This has to do with hardware limitations of the memory system: For modern processors, it is slower than the processor I/O rate by order of several magnitudes. Fast, but small local memory (*cache*) has been introduced to cope with this. Additionally, modern processors are internally super-scalar pipelines, that offer SIMD instructions for working in parallel. For good numerical performance, this reality must be honored and used consequently.

The bad scalability effects of memory access will continue to get worse — the gap between processor speed, memory bandwidth and memory latency is widening. Sophisticated programming in high level languages like C++ can only provide algorithmically efficient programs. For numerical performance, hardware features must be honored. Although compilers will learn to make use of special processor instruction sets, they will always be one step behind current technology. Speaking more aggressively: Tuning C++ code is no excuse for ignoring SIMD processing options.

G
page 145

In this chapter, we also review some hardware features that are essential for interactive computer graphics and present "custom programmable" integrated circuits for general use. They all have in common to efficiently take work load off the processor. When designing interactive applications, it is crucial to employ them as a fine parallelization

and distribution strategy that share the work between them and the main processor. As they are working independently and asynchronously, communication bandwidth and latency considerations play an important role in such a design.

3.2 SIMD processing

Modern microprocessors offer special purpose extensions to their standard instruction set. The most prominent ones are MMX, SSE and SSE2 [Int00] of Intel, 3DNow! and 3DNow!Ext [AMD00] of AMD and AltiVec for SPARC processors [Mot99]. The first extension that has been proposed was MMX, aiming at accelerating multimedia applications (bitmap processing). It has instructions for executing up to 4 integer operations as SIMD at the same time. It is nowadays largely supported by nearly any Intel x86 compatible processor. For floating point SIMD instructions, Intel and AMD created different incompatible instruction sets, SSE and 3DNow!. They have been superceded by extended versions SSE2 and 3DNow!Ext.

G
page 145

When it comes to numerical performance, modern computer architectures exhibit a serious bottleneck: Access to main memory. Especially in the PC architecture, processor speed gains over the last years have exceeded the performance gain of memory chips by far. The cure is to introduce multiple levels of high speed caches that tend to get bigger with every processor generation. The rationale therefore is, that good performance can only be achieved with algorithms that honor existing caches. Writing cache–aware algorithms however is a highly complex topic of its own.

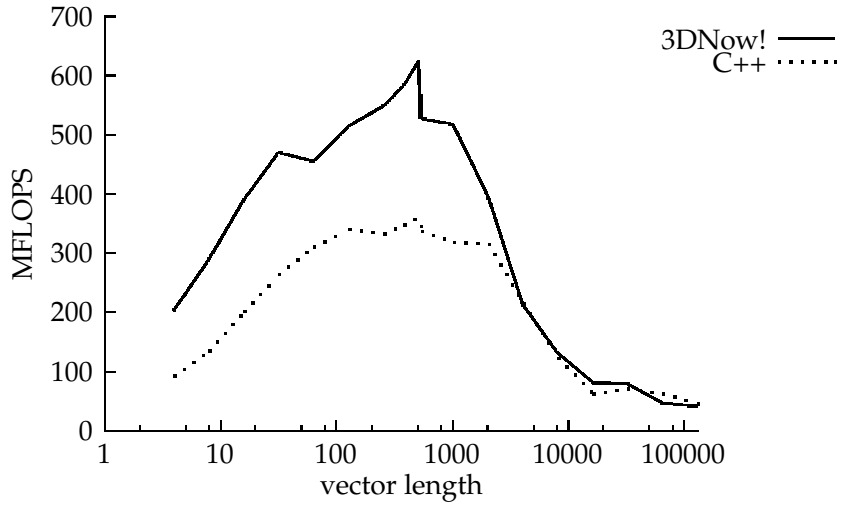


Figure 3.1: Performance gain when using the SIMD instructions of modern processors for in–cache algorithms.

In the context of parallelization strategies, we examine in this section the possibility of using the processor caches for high numerical performance by using SIMD algo-

rithms for in–cache data. We use the 3DNow! extension of AMD that offers SIMD floating point instructions. They allow the processor to generate up to four 32-Bit, single-precision floating point results per clock cycle [AMD02].

The goal of this section is to show the importance of first accelerating the numerical operations, so that the program becomes purely memory-bound (→ figure 3). Then, it makes sense to alter the algorithmic design of the program in the high level language to ensure data locality. Attacking the problem on the high level first can give suboptimal results, because performance measurements will incorporate the cache effects which will change if SIMD processor instructions are introduced afterwards.

All following experiments were run on an Athlon 800 processor. Figure 3.1 displays the characteristic curve for memory-bound algorithms. After an initial rise, the performance suddenly breaks down as soon as the data does not fit into the cache anymore. The interesting point however is here, that with using SIMD instructions, the performance can nearly be doubled. This is due to less processor instructions that have to be issued in the SIMD case. In figure 3.1, we simply add two vectors of varying length.

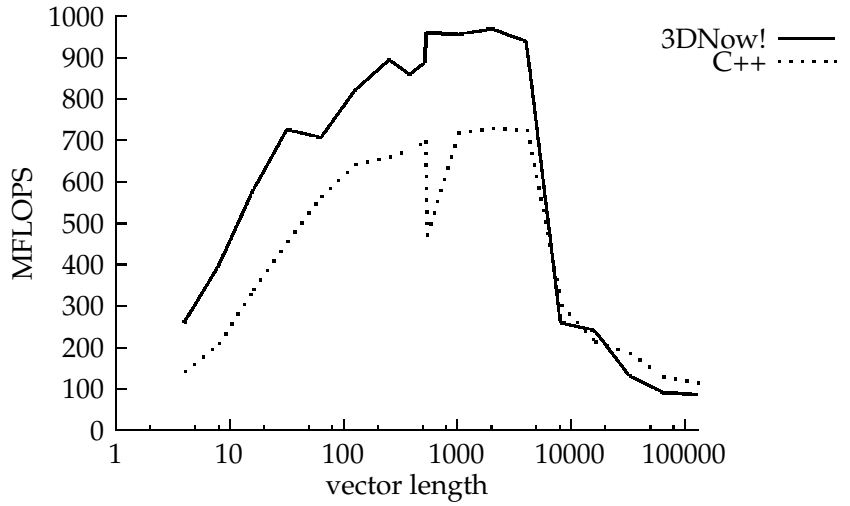


Figure 3.2: Using SIMD instructions to compute the dot product of two vectors.

One drawback of the SIMD approach is, that most high level compilers currently do not support the extended instructions sets for 3DNow! or SSE. Therefore, efficient use of SIMD instructions is only granted when programming in assembly language. Fortunately, the operations for which SIMD instructions can be used with reasonable effort, occur only in very small parts of the program. Writing these parts in assembly language usually results in very few lines of code. When these code blocks are wrapped within inline C++ class methods, the SIMD performance can be accessed very conveniently and efficiently. The code in listing 3.1 is an example of this approach: It computes the dot product (scalar product) of two vectors of arbitrary length by employing 3DNow! instructions, whose mnemonic is prefixed with `pf`. The basic idea for getting maximum performance is, to process the vector components in groups of four values using the

SIMD registers `%%mm0` to `%%mm5`. If the length of the vectors (traced in the `%%ecx` register) is less than four, process them in groups of two. If it is less than two, do it separately. Pointers to the two vectors are kept in the `%%eax` and `%%edi` register. After each step, the intermediate results are summed up and the final result is returned in the `%%edx` register.

Figure 3.2 shows the performance of the code in listing 3.1 for two vectors of varying length. Obviously, both the C++ version and the 3DNow! version now give better performance than in the first experiment. This is because the add operation of the first experiment was just too simple and the algorithm therefore was still memory-bound. This shows the importance of careful loop design, if maximum performance is a critical point. Using SIMD instructions therefore is most effective, if the algorithm is computationally expensive. In figure 3.3 we have performed a component-wise square root computation on vectors of varying length. The FLOPS rates here are an estimation, as the true number of floating point operations, the `sqrt` function performs is not known. But the principle statement remains valid, that when using SIMD, we are able to efficiently use the cache. The standard C++ version that uses the `sqrt` math library call is already slower than accessing main memory and therefore exhibits almost no performance gain for vectors that fit in the cache.

The SIMD code fragment for the square root computation is even smaller than the code for the dot product, as 3DNow! offers a `pfrsqrt` call. The instruction set has many more efficient SIMD instructions for division, inversion and inverse square root computation of floating point numbers.

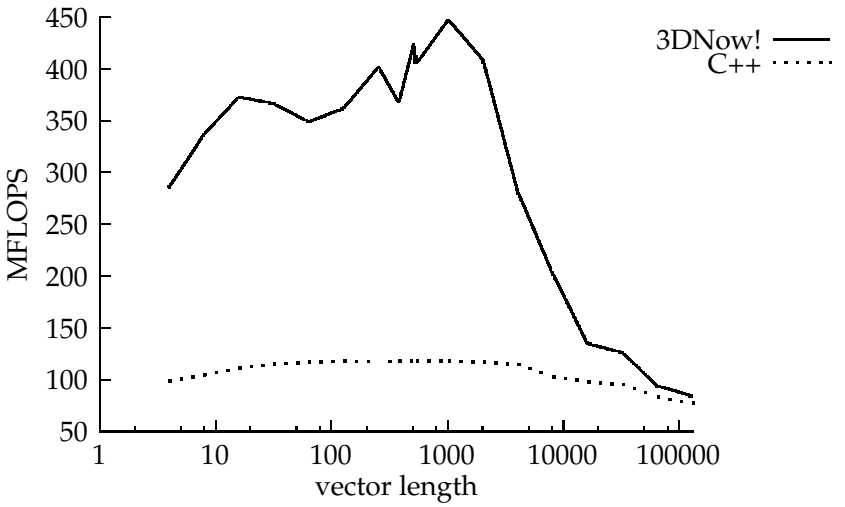


Figure 3.3: Using SIMD instructions to compute the component-wise square root of a vector.

The use of SIMD processor instructions is a very valuable fine grained parallelization strategy. Here is a short summary:

PRO/CON

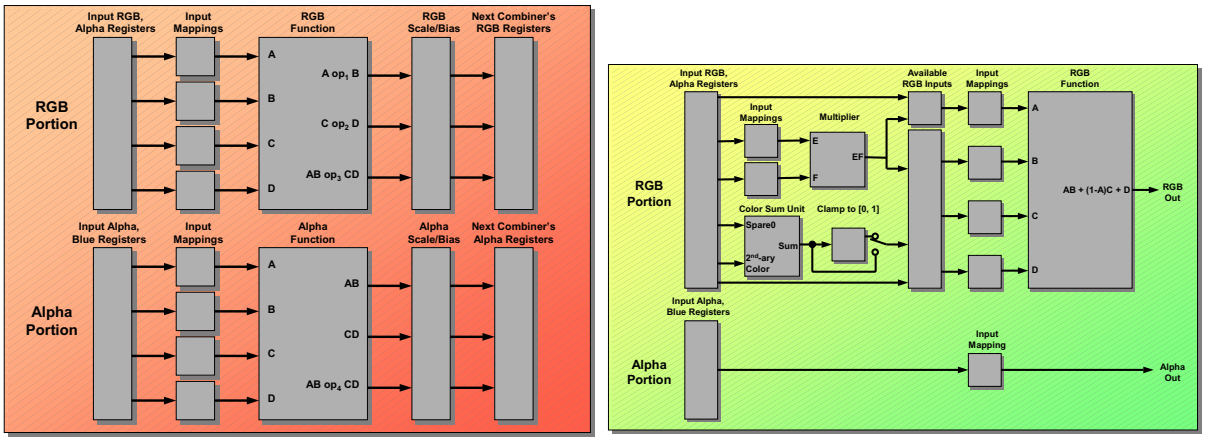
- ✓ It can accelerate computationally inexpensive operations like addition, because of a lower number of processor instructions to be scheduled.
- ✓ It does not degrade performance if the data does not fit in the cache.
- ✓ It allows to exploit cache performance for computationally expensive operations that would otherwise take longer than main memory access time, i.e. it pushes the limit at which an algorithm becomes memory-bound.
- ✗ SIMD algorithms have to be programmed in assembly language, if careful control is necessary or instruction set is not supported by the high level language compiler.

3.3 Custom hardware programming and design

For interactive applications, making efficient use of given hardware features is imperative. Desktop computers, like PCs or graphics workstations usually have some hardware devices that perform visualization and rendering related tasks independently of the processor. The most common device of course is the graphics card that implements a frame buffer and provides 2D drawing routines in hardware. As these 2D operations are very simple to implement, they have been there since the introduction of pixel-oriented CRT displays.

For special tasks, many graphics boards offer advanced 2D operations like motion compensation for MPEG movies, capture and overlay pixel planes for video display and editing, or even complete video codecs in hardware. For time-critical tasks like video and animation, the visual quality usually drops dramatically when the tasks the hardware can perform, are executed on the processor. Interestingly, this is most often not due to numerical performance problems, but rather because of the data size to handle or latency issues (dropped frames, bus saturation, interrupts). The observation is, that nearly all the operations performed are embarrassingly parallel and numerically simple at the same time. It is therefore clear, that one should strive to do all numerical intensive pre-processing on the CPU and employ hardware support for 2D mass operations afterwards.

Following this strategy, graphics cards that support drawing 3D objects have been implemented. They range from complete and expensive hardware implementations of the OpenGL pipeline on graphics workstations to cheap PC graphics cards, that only perform texturing and lighting in hardware. The main challenge here is the combinatorial explosion of possible rendering modes unlike the simple 2D operations. The solution is to create *programmable hardware* implementations. For PC graphics cards, this trend has also enabled to perform some of the geometry handling in hardware.



(a) (b)

Figure 3.4: The *n*VIDIA general combiner (a) and the final combiner (b) can be configured for texture operations and fog. Figures taken from [NVb]

Some of the implementations that are discussed in the third part of this thesis employ the 3D capabilities of *n*VIDIA PC graphics cards. For improved visualization and rendering, the *register combiners* [NVb] have been used. They provide per-pixel operations that can be assembled from fixed circuit blocks that are invoked once per pixel. The actual operation each block performs and the interconnect of the blocks can be customized by an OpenGL programming extension. It allows sophisticated lighting and shading without a frame rate penalty. The second *n*VIDIA extension used is the *vertex shader* [NVc]. It gets invoked once for every geometry vertex that is processed. The shader is specified as a small assembler like program that is compiled and executed on the graphics card. The small instruction set contains all basic operations that are frequently used in graphics algorithms like addition, multiplication and the dot product. As the vertex shader has access to geometry coordinates, it can be used for advanced processing other than lighting calculations.

Recent developments of graphics boards however show a clear tendency towards *custom programmable hardware*. The *n*VIDIA vertex shaders are only a first step. The possible applications of completely free programmable graphics co-processors are endless, ranging from special effects for games to high quality shading and rendering. Circuits with access to local graphics memory and dedicated arithmetic units would even allow to implement different rendering methods like ray tracing or global illumination methods. The results may be used to complement the triangle rasterization or even replace it completely.

G
page 145

A very promising approach to inexpensive custom hardware for graphics is to use *field programmable gate arrays (FPGA)*. They belong to the group of *semi-custom de-*

sign style programmable integrated circuits, as opposed to *full-custom design* style circuits like *application specific integrated circuits (ASICs)*. The design is called semi-custom, because the circuit already contains a large amount of fixed logic and switching parts and possibly multiple layers of interconnects. The *personalization* of the circuit, i.e. the programming, is done by uploading a bit-image to a configuration matrix field. The matrix entries represent a one-to-one mapping of all active parts of the integrated circuit. By enabling a matrix entry, the connection between the two corresponding parts is set. In doing so, a program can be built that uses the given active parts. The matrix can be reconfigured and the circuit therefore can be reprogrammed. Because the circuit is personalized by the user, the hardware manufacturing costs are very low.

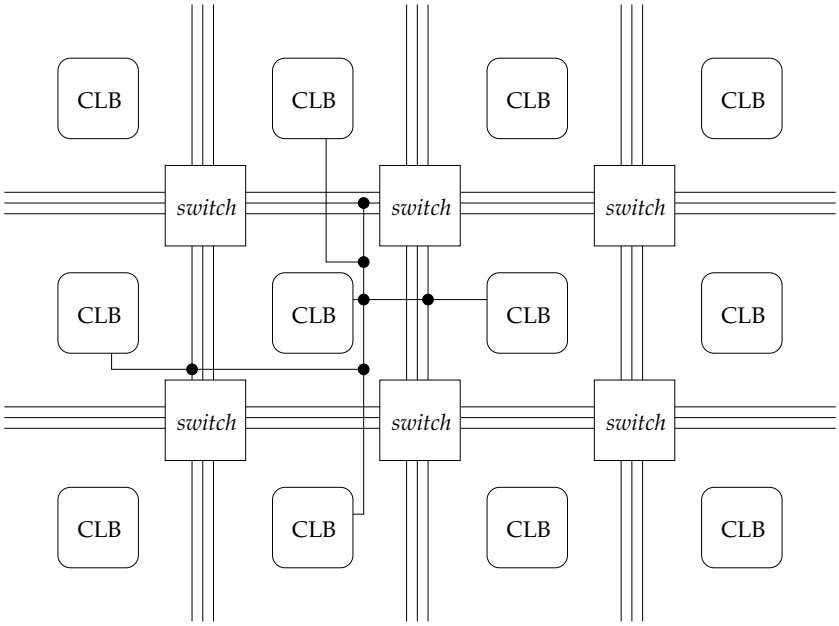


Figure 3.5: Internal layout of CLBs and switches on a Xilinx Virtex FPGA circuit.

The active parts of a FPGA are called *configurable logic blocks (CLB)*. They usually implement some very simple logic operation and host multiplexers and interconnect terminals. For building more complex logic functions, several CLBs have to be connected. The interconnects available on the circuit normally do not allow connecting arbitrary CLBs. They are organized hierarchically and are fragmented in local and global connects. The FPGA design therefore suffers from slicing. The connection hierarchy varies with the manufacturer. For storing intermediate results, local memory registers are available on the chip. Specialized I/O registers connect the FPGA to the outside world via the pins of the chip.

The principal FPGA technology is shown in figure 3.5. This layout is implemented within the Virtex circuits of Xilinx [Xil01a]. The CLBs are organized as a matrix that is surrounded by I/O blocks. There are horizontal and vertical busses to connect distant CLBs. Each CLB has an associated switch for accessing these busses. In addition, each

CLB can contact the adjacent CLBs directly which minimizes the envelope delay. This allows tight coupling and long range connections as needed by the personalization.

3.4 Discussion

Although the SIMD processing and the FPGA strategy presented in this chapter do not compete directly, it is instructive to have a comparative look at them. They implement very different approaches to data processing.

To illustrate this, we implement integer vector addition as an example. Listing 3.2 shows the assembler code of a fictitious machine that can operate on address references: If the address is not preceeded by a hash mark (#), the memory content at the location that is stored in the given address is loaded into the accumulator. The setup for the example code expects the address of the first vector in memory location `[0]`, the address of the second vector in `[1]` and the address for the result in `[2]`. Memory location `[3]` holds the loop counter. The FPGA hardware implementation is shown in figure 3.6(a). It feeds the vector components directly into an adder that emits the result to the third vector. The addresses of the vector components are generated by a loop counter that is incremented synchronous with the adder in every clock cycle.

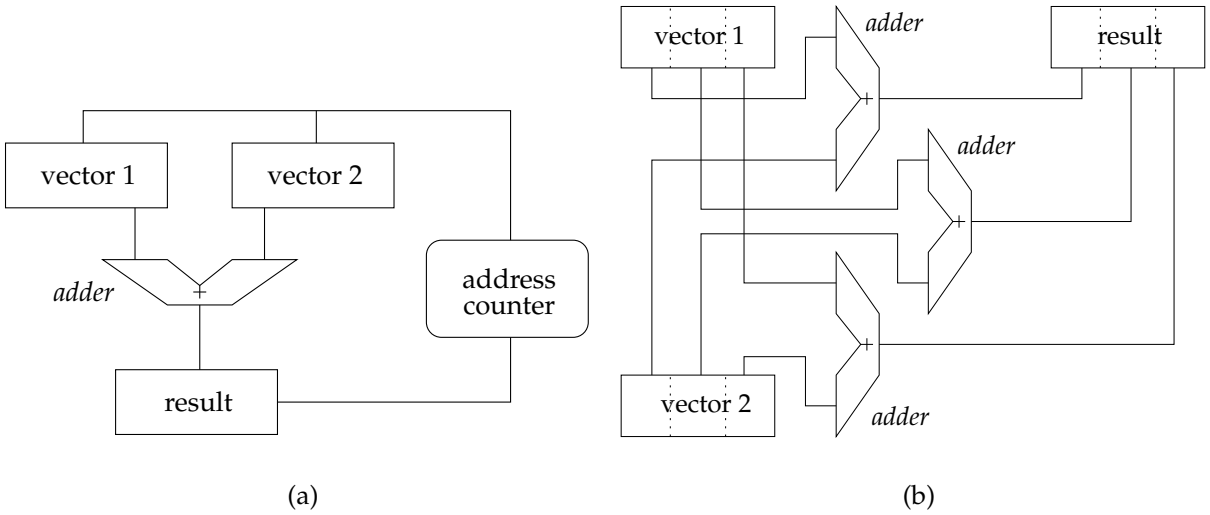


Figure 3.6: Block diagrams for adding two integer vectors on FPGA hardware.

For a comparison, we assume that the processor has at least a three stage pipeline (instruction fetching, decoding, executing), caches and an optimal branch prediction unit so it can execute every instruction in a single clock cycle. For the FPGA hardware, we assume that memory, adder and counter are on the same chip, so one addition can be performed per clock cycle. In the following, we neglect pipeline setup and initialization times.

G
page 145

When adding two three-dimensional vectors, the software solution needs 48 clock cycles, the FPGA hardware only 3. This represents a speedup factor of 16 for the hardware solution. The software can be tuned for fixed vector lengths by manually unrolling the loop. Then, the software would need 36 clock cycles. For the FPGA hardware, figure 3.6(b) shows that one can do exactly the same by replicating the entire subsystem, so everything will be done in a single clock cycle. This represents a speedup factor of 36 ! The factor may get smaller if SIMD instructions are available on the CPU.

The example clearly shows the theoretical capabilities of the FPGA technology that a program can benefit from. But for a fair comment, one has to take into account, that

- nowadays processors (ASIC technology !) operate at frequencies that are a factor of 20 higher than with FPGAs.
- we have considered adding integer values. Executing floating point operations in one clock cycle will produce very deep FPGA logic, which limits the maximum achievable clock rate. Amortized single clock cycle floating point operations are no problem for modern CPUs.
- we have performed the addition operation. Implementing a multiplier on the FPGA demands a huge number of logic gates and thus such algorithms are limited by the available chip surface. For modern CPUs, any basic arithmetic operation is equally fast.

Facts

The overall rationale is, that one has to decide carefully whether to use FPGA technology. Its biggest drawback however is the lack of availability on consumer machines. However, when a large number of relatively simple operations on a large amount of data has to be performed, FPGAs can deliver massively parallel processing facilities that can outperform the CPU by far for specialized tasks.

```
  pxor %%mm4, %%mm4     // clear registers
  pxor %%mm5, %%mm5
  sub $4, %%ecx         // check if length less than 4
  jb mul4_skip

mul4_loop:              // do 4 values at a time

  movq (%%eax), %%mm0
  movq (%%edi), %%mm1
  movq 8(%%eax), %%mm2
  movq 8(%%edi), %%mm3
  add $16, %%eax
  add $16, %%edi
  pfmul %%mm0, %%mm1
  pfmul %%mm2, %%mm3
  pfadd %%mm1, %%mm4
  pfadd %%mm3, %%mm5
  sub $4,  %%ecx
  jae mul4_loop         // check if length greater 4
  pfadd %%mm5,%%mm4

mul4_skip:

  add $2, %%ecx         // check if length greater 2
  jae mul2_skip
  movq (%%eax), %%mm0   // do 2 values at a time
  movq (%%edi), %%mm1
  add $8, %%eax
  add $8, %%edi
  pfmul %%mm0, %%mm1
  pfadd %%mm1, %%mm4

mul2_skip:

  and $1, %%ecx         // anything left ?
  jz even
  pxor %%mm0, %%mm0     // yes, uneven length
  pxor %%mm1, %%mm1
  movd (%%eax), %%mm0   // combine results
  movd (%%edi), %%mm1
  pfmul %%mm0, %%mm1
  pfadd %%mm1, %%mm4

even:

  pxor %%mm5, %%mm5     // no, even length
  pfacc %%mm5, %%mm4
  movq %%mm4, (%%edx)
```

Listing 3.1: x86 assembler routine for computing the dot product using 3DNow! SIMD instructions.

```
loop: LOAD 0    // ACC = [0]
      ADD 1     // ACC = [0] + [1]
      STORE 2   // [2] = ACC add vector elements

      LOAD #0   // ACC = [0]
      ADD #1    // ACC += 1
      STORE #0  // [0] = ACC increment address vector 1

      LOAD #1   // ACC = [1]
      ADD #1    // ACC += 1
      STORE #1  // [1] = ACC increment address vector 2

      LOAD #2   // ACC = [2]
      ADD #1    // ACC += 1
      STORE #2  // [2] = ACC increment address result

      LOAD #3   // ACC = [3]
      SUB #1    // ACC -= 1
      STORE #3  // [3] = ACC decrement loop counter

      JNZ loop
```

Listing 3.2: Assembler code for adding two integer vectors.

Part II

Integrating Simulation, Visualization and Rendering

4

THE *gridlib* PROJECT

Man muß etwas Neues machen,
um etwas Neues zu sehen.

— *Georg Christoph Lichtenberg*

4.1 Introduction

The goal of the *gridlib* project is to develop a modern object-oriented software infrastructure for common grid-based numerical simulation problems on trans-teraflops machines. These supercomputers and modern scalable algorithms allow *numerical simulations* to be performed at unprecedented grid resolutions. However, this also tremendously increases the sizes of the result data sets, surpassing the capabilities of current pre- and post-processing tools by far. At the same time, pre- and post-processing has become more and more important. Current complex engineering solutions require the automatic generation of problem-specific, time-dependent, adaptive, hybrid 3D grids that can be partitioned for parallel simulation codes. Enormous amounts of data must be presented visually for easy interpretation.

The system hardware of current supercomputers also places non-trivial demands on the software architecture, in particular the widening gap between the low bandwidth of external communication channels and the available size of local memory. This requires the execution of pre- and post-processing steps on the supercomputer, which is a significant problem due to missing generic software support. Other difficulties arise since only special data and software structures can be efficiently handled on the high performance architectures. A naive implementation may lead to unacceptable performance problems.

G page 145

The *gridlib* project addresses these problems. It can act as a *middleware* between existing software modules for pre- and post-processing. It furthermore allows for implementing efficient solvers for complex simulation tasks. In this chapter, we present an

architectural overview of the *gridlib*, describe its current functionality and show some example applications.

4.2 Overview

The *gridlib* architecture provides three major *abstraction layers* (→ figure 4.1) [Kip00, Kip01b]. The lowest one is responsible for encapsulating the actual memory layout of data. Because the next layers entirely rely on this abstraction, the lowest layer can organize the storage freely. In particular, it can format its own memory layout to conform to the memory layout of other third party codes. We can exploit this possibility for using a binary-only third-party flow solver.

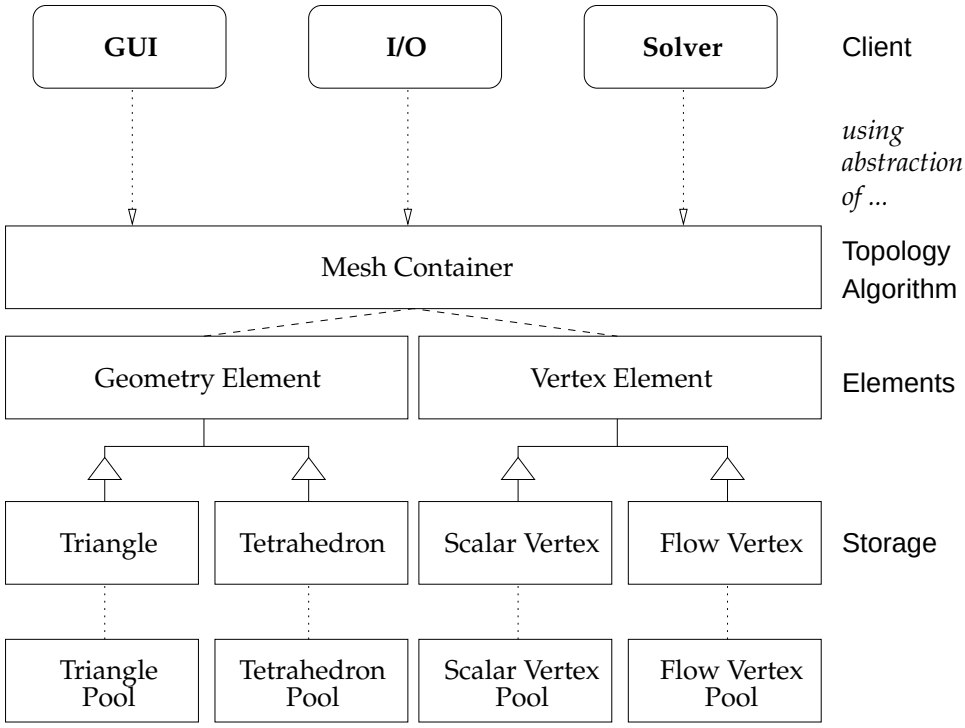


Figure 4.1: The *gridlib* provides three major abstraction layers for both integration of binary-only codes and development of new object-oriented solvers.

The second abstraction layer is the main link to the object-oriented world. It provides interfaces for all primitive elements (triangles, quadrilaterals, tetrahedra, hexahedra, prisms, pyramids, octahedra), edges and vertices as regular C++ classes. This sets the *gridlib* apart from other grid management libraries, as most of them do not allow the programmer to actually call methods on these objects.

The topmost layer provides the concept of a mesh container. It does not make any

assumption on the mesh topology and implements abstract services, like neighborhood setup, subdivision functionality and management and content iterators.

The mesh container and the element abstraction layer provide powerful object-oriented programming support. For the library user, the *gridlib* further implements several clients that use the three-tier architecture for disk I/O, visualization and simulation. The performance of the interfaces for the data exchange between the grid management, the solver and the visualization and rendering subsystems has been evaluated by performing several simulations in the context of the KONWIHR project "*gridlib*: A parallel, object-oriented framework for hierarchical-hybrid grid structures in technical simulation and scientific visualization" [KG01, KHM+02].

4.3 Storage abstraction layer

In order to provide abstraction with respect to the memory layout of storage space, the data of each primitive object type is taken care of by a *memory pool*. Each pool implements a specific storage layout. In general, each primitive object type is associated with a specific memory pool implementation. This does not seem to provide great flexibility, but it has a number of advantages that are discussed below.

Because of the very concrete nature of each memory pool implementation, there is need for introduction of an additional concept for achieving all the benefits object-oriented languages are renowned for. Although all memory pool implementations share a common functionality, because of the differences in memory layout and for efficient data packing, using inheritance is prohibitive. However, dealing with many unrelated object types is very bad programming practice and does not support abstractions, which in turn is vital for re-usability and maintenance of the code. Section 4.4.1 therefore introduces a design pattern that has been used to solve this problem.

G page 145

4.3.1 Memory Pools

The *gridlib* offers a memory management subsystem that is able to store small objects in a memory pool [Kip00]. Each pool defines a specific storage layout used by a primitive object. The pool functionality can be described in a generic way. The class `GbMemPool<T>` provides a chunk of memory that can be acquired by a client in arbitrary granularity. It is put back into the pool for reuse when the client releases the memory (→ figure 4.3). This concept has several advantages and some minor drawbacks:

PRO / CON

- ✓ Because there is no inheritance and abstraction involved, each pool implementation is guaranteed to only require storage for its data members.
- ✓ The absence of a `vtbl` also ensures that `inline` instructions are working.

- ✓ Because `GbMemPool<T>` only has to manipulate a few pointers for each request, it is much faster than individual calls to the operator `new` or the operator `delete`.
- ✓ The size of the memory chunks allocated by the pool can be optimized to fit system properties (`int`-alignment, page size, ...).
- ✓ To the pool's functionality, the layout of the individual data record to be stored is of no importance. The pool can therefore be implemented as a template that is parameterized over the memory layout declaration.
- ✗ The minimum size of an individual item from which a pool is to be constructed is the size of a pointer, which is 4 bytes on most systems. The pool concept implemented here therefore does not work for pooling single byte `chars`.
- ✗ The pool can only handle objects of the same size.
- ✗ Deriving from an existing pool declaration makes no sense, because it destroys the terminal-class property of the pool. For creating several pools in one application, each pool has to be declared from scratch.

Typical applications of a pool of `chars` or `shorts` occur frequently in time-critical algorithms, which have very strict requirements on timing and storage complexity. There exist a number of highly specialized solutions for this group of problems. The `GbMemPool<T>` class does not try to break into this domain and concentrates on the more general parameterized pool concept. Finally, the drawback of not being able to derive from a memory pool seems to be the most severe. As the memory pool in *gridlib* is implemented as a template class, deriving from it is not necessary. Nevertheless, handling all pool instances in a way as if they have a common ancestor is possible by employing an additional design pattern that is presented in the next section.

4.4 Element abstraction layer

One of the main aspects of the *gridlib* framework is to provide real object semantics to the programmer. This includes well defined interfaces for the participating object groups like vertices, geometry elements and edges. For each group a (pure virtual) interface class is provided along with several derived classes that implement standard object types, like a triangle. In the previous section, efficient storage of small objects has been discussed. The developed memory pool concept is now used for implementing the basic vertex, edge and geometry objects of the *gridlib* [Kip00, GKLT00].

4.4.1 External Polymorphism

For the implementation of the basic objects, the *gridlib* uses the concept of *external polymorphism* [CS98, GHJV95] to satisfy the demands for abstraction and flexibility. It

is implemented in a subsystem of wrapper classes that link to some appropriate memory pool. The wrapper classes themselves now being totally decoupled from the storage layout, can employ every object-oriented pattern possible, and act as *proxy interfaces*. In particular, they define inheritance relations of primitive objects by declaring common pure virtual abstract interfaces. This is the fundamental key to code maintenance and algorithmic design on higher levels.

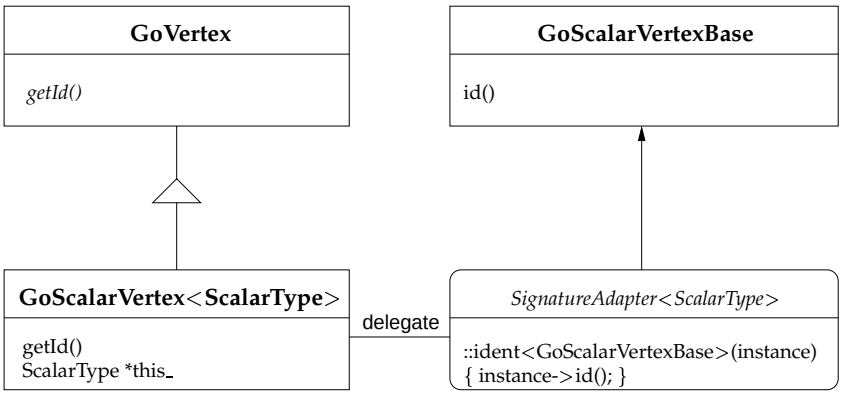


Figure 4.2: UML object-model of the external polymorphism design pattern.

Figure 4.2 displays a UML diagram of the external polymorphism pattern. The goal is to create an inheritance relation for several distinct classes, like the **GoScalarVertexBase** class on the upper right. We may not simply add a pure virtual interface and derive the classes from it, as they may not have the same interface or direct inheritance is prohibited because of performance reasons. But as our goal is to have a common interface, we declare a pure virtual one that suits our needs: **GoVertex** on the upper left. Now we derive from it one class (**GoScalarVertex** in our example) for every basic class we want to reach. The methods of this class simply delegate the work to sufficiently global signature adapters that are parameterized with the destination class's type. The signature adapter now calls the destination method with the correct name and parameters. Note that the signature adapters are template functions and therefore will be specialized by the compiler automatically. The outcome of this procedure is a full inheritance hierarchy for totally unrelated class implementations.

Here is a short discussion of the external polymorphism design pattern:

PRO / CON

✓ **Low overhead:** In a conventional class hierarchy, derived classes may inherit an unnecessarily large interface. For consistent code changes, the source code must be available: If concrete data types must be extended, their memory layout must be adapted through virtual pointer tables, which is prohibitive for some libraries like STL. When using the design pattern, this is not a problem because the memory layout is delegated to some concrete class.

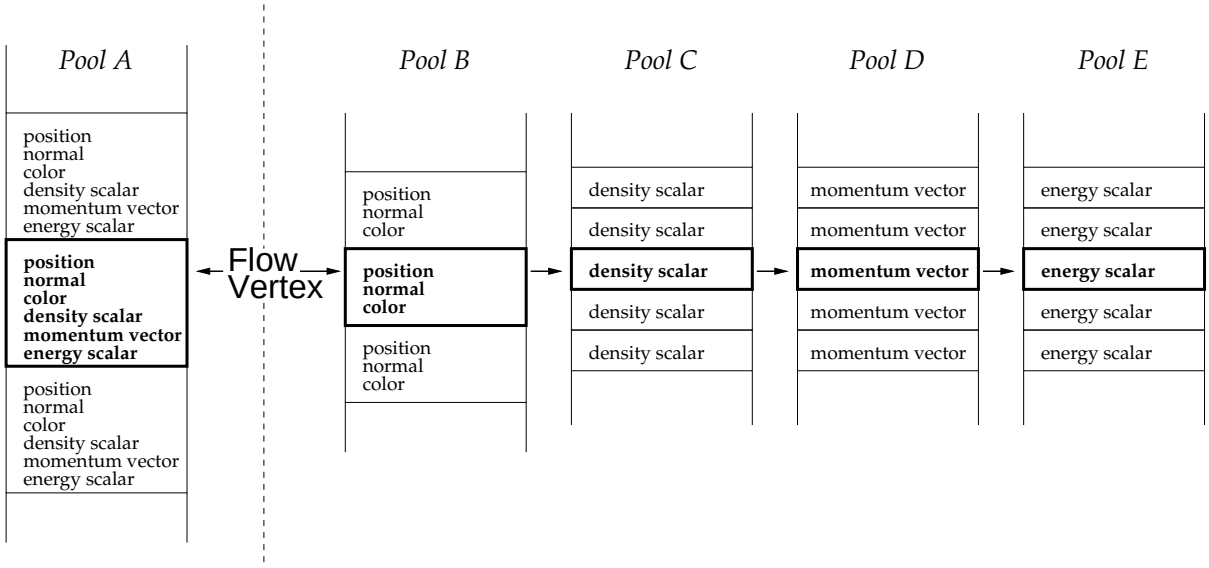


Figure 4.3: The storage for primitive objects can be allocated from a single memory pool (left side) or employ a composition of several pools (right side) according to the expectations of the solver code.

- ✓ **Universal:** Given an object system, in which the participating classes must register their methods explicitly (via static type coding / brute force casts), every combination of classes and libraries must be globally encoded manually. This design pattern in contrast only needs changes to happen in the adapter implementation which is highly localized code.
- ✓ **Transparent:** Classes that don't collaborate conceptually can be treated polymorph with respect to the virtual interface. In particular, changes in the virtual interface do not require changing the original objects.
- ✓ **Flexible:** Using a programming language that supports parameterizable types (templates), even primitive data types like `int` or `float` can be treated polymorph.
- ✓ **Peripheral:** Because the design pattern does not directly interfere with the objects, it can be entirely removed by conditional compilation, if its functionality is not useful (debug-only functions).
- ✗ **Unstable:** The access adapters and the virtual interface must be changed accordingly, if the interface of the original objects is changed.
- ✗ **Obtrusive:** It must be examined carefully, if the code using the design pattern must use the virtual interface or must call the objects directly. Adding the design pattern to existing source code can incur large code changes.

✗ **Inefficient:** The pattern can introduce several levels of virtual method dispatching, if the original objects are not able to provide `inline` methods, in which case there is only a single dispatch necessary.

✗ **Inconsistent:** Because the original objects *implement* the pseudo-polymorph methods indirectly but do not *declare* them directly, the methods are not reachable using a pointer to the original object.

The *gridlib* declares abstract interfaces for each logical group of *primitive objects*: `GoGeometryElement<T>` for geometric primitives like triangles or tetrahedra, `GoEdge<T>` for edges and `GoVertex<T>` for nodes. Each primitive object that is derived from these interfaces implements a wrapper for one or more memory pools (→ section 4.3.1). The flexibility of this approach is, that such an implementation can use any number of memory pools to store its data. In some application, it may be appropriate to put all data into a single pool. If some subsystem implemented in a programming language like FORTRAN is to be used, it may be more appropriate to employ a distinct memory pool for each value to be stored. In figure 4.3 two possibilities for implementing a flow vertex are displayed: On the left side, all data is allocated as a single block from a memory pool, while on the right side, the data is stored in three specialized memory pools. This is the exact analogue to the decision of using an array of structs or a struct of arrays for data storage.

G page 145

All primitive objects are derived from the abstract interfaces and internally take care of the link to the memory pool. They appear to the programmer like standard C++ objects. This includes in particular object creation / destruction and the object hierarchy available, enabling the writing of generic algorithms that use the parents virtual interface. Appendix C has an overview of the inheritance relations for all core interfaces of the *gridlib*.

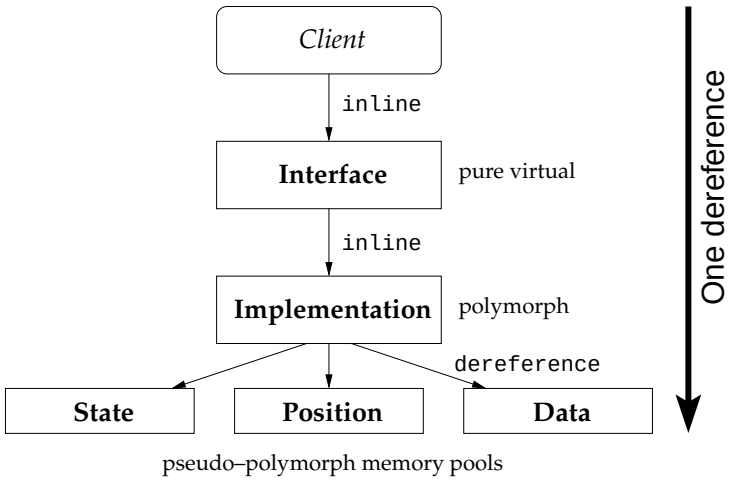


Figure 4.4: Accessing the pseudo–polymorph memory pool implementations reduces to a single dereferencing operation.

Because inheritance relations are defined by the wrapper objects and not within the implementation of the basic memory pooling classes, the declaration of polymorphism is outside of the scope of the pool class's declaration (→ figure 4.2). This is why the design pattern is called *external polymorphism*. The wrapper classes can choose freely how to apply a combination of *decorator* and *substitution* pattern

onto this. Because the linked class implementations (the memory pools in our case) need not be related by inheritance, there is a chance that inlining is working. In fact, as shown in figure 4.4, with the design of the *gridlib* this is the case. Consequently, a call to a virtual method of the wrapper objects requires just a single dereferencing operation and therefore is as expensive as any virtual method call. In other words, the external polymorphism pattern does not introduce a run-time penalty compared to conventional implementations that directly derive objects from a common interface to provide real element semantics. The *gridlib* uses this design pattern to decouple the memory layout from the element abstraction.

4.5 Mesh abstraction layer

As shown in figure 4.1, the uppermost system level provides abstraction of a grid through a container. It is responsible for encoding topology and geometry of a mesh by exclusively using the abstract element interfaces below. Therefore, the mesh algorithms work on any given storage layout. The mesh acts as a pure container, which means that its algorithms may not make any assumption about the implementation of the elements or the grid topology. The mesh simply acts as a collection of basic elements (geometry elements, edges, vertices) that form a logical entity in space. A mesh can have an arbitrary number of levels of subdivision [Kip01a].

G page 145

The mesh also provides some functionality on the set of elements it contains. Most of the methods are query methods for topological context or quantitative properties. There are also some service routines for the convenience of clients. As the mesh layer is the most important interface to the clients, appendix D lists the provided capabilities.

4.5.1 Algorithmic abstraction

The second important concept of the *gridlib* is the efficient formulation of algorithms on the mesh level. The mesh abstraction layer provides *algorithmic abstraction* to clients. It allows efficient access to the contained elements and promotes a programming style, that makes formulating client algorithms very maintainable.

Practically every algorithm dealing with a grid performs `foralll` iterations on its elements, vertices or edges. The *gridlib* therefore provides a concept that supports this particular algorithmic pattern. One can write small code entities, called *functor*, that are to be applied to all elements, vertices or edges of the mesh. The mesh has methods that take any such functor definition, construct the functor and apply it to every contained element, vertex or edge. Because a functor uses the abstract element interfaces below, one can use the full possibilities of overloading, inheritance and runtime type information when writing one.

G page 145

A functor is implemented by overloading the default operator `operator()` of a lightweight class. The mesh container provides iterators for executing the functor on

every grid element. Note that this concept is very useful for parallelization because of the clear separation and encapsulation of the working domain of the functor: Depending on the point in time when a functor is constructed, its private data members reside in shared or thread-specific memory. The actual algorithm that is formulated as a sequence of functor applications to the mesh is called *skeleton program*.

The pseudo-code example in listing 4.1 sets a marker flag for every quadrilateral in the mesh and calls method `foo()` on all objects of the mesh. For vertices, the `fooSpecial()` must be called with a parameter. In terms of object-oriented programming beauty, the obtained code is of high quality and the interacting code entities are clearly distinguishable. Here is a short discussion of the concept:

PRO / CON

- ✓ **Flexible:** The concept easily handles multiple base classes and objects with different method names and signatures.
- ✓ **Robust:** If a new object type is derived from the class interface that is the functors argument, the skeleton program behaves very sensible by using the method with the parent signature.
- ✓ **Localizing:** If, for some reason, the `foo()` method must perform an additional task on all or on some specific objects, it is clear where the source code to be modified is located: `MakeFoo::operator()`. This tremendously helps the program maintenance and promotes code reuse.
- ✓ **Understandable:** The modularity of the functor usually results in small code blocks inside the `operator()`. Additionally, the operator typically has only a few arguments, making the code block tight and easy to understand, because irrelevant pre- and postconditions are not visible.
- ✓ **The big picture:** If there is a reasonable functor granularity and little helpers like `m.forAllObjects()`, the skeleton program is very clear and easy to understand. All detail is taken care of by the functors implementation. The skeleton therefore can concentrate on formulating high level algorithms.
- ✓ **Safe:** Because of the limited actions a functor performs, the possible negative side-effects it can produce are minimized. The functor parameterization also enforces additional type checking on the arguments.

Note that a functor must be "sufficiently" global with respect to the skeleton algorithm. A functor therefore can be private to a specific class, can be inherited and serve a whole subtree or can be absolutely global to provide for example a conditional `delete` call on the abstract top-level interface to remove all marked elements, vertices or edges. Putting some generally useful functors into the global scope provides powerful support for code reuse of skeleton algorithms on the mesh level. It is also a good means of providing library functionality to developers.

Last but not least, the code locality, the clear definition of data storage and data access and the restricted scope of the operations also helps the optimizer engines of current C++ compilers to better figure out whether aliasing and dependency relations prohibit optimization or not.

4.6 Clients

When writing applications that deal with meshes, several tasks occur routinely that are not trivial to implement, but have at the same time nothing to do with the actual computation. The classic example are input and output routines that must be portable (endianess, file system), fast (possibly parallel) and easy to use. The *gridlib* provides such services as library functionality that is implemented as modular subsystems that access the core grid routines exclusively through the high level interfaces on the mesh and element level (→ figure 4.1). The services are therefore completely independent of any application specific topics. As all subsystems have well defined interfaces and encapsulate the functionality well, each one is provided as a small auxiliary dynamic library. It needs only to be linked to an application if its functionality is accessed. When implementing a program that uses the core *gridlib* routines, one can therefore decide freely to re-implement critical functionality, or use the provided services for rapid development.

4.6.1 Services and Utilities

As all library services access the grid by the high level interfaces, they are considered to be clients to the *gridlib*. The actual application then acts as a client to the service. The services provided by these subsystems are diverse and cover a wide range of topics. With the exception of the visualization and the rendering subsystems, which are discussed in detail in the next section, the following list gives a short overview of the implemented functionality, as the gory details are simply off topic of this thesis. Note however that without this functionality, successful implementation of the applications presented in the third part of this thesis would not have been possible.

- **Util:** This subsystem provides a collection of little programming tools of general usefulness. It is very likely to be used in every application: One- and two-dimensional Bit arrays, 2-Bit arrays, binary trees, an image class of configurable color format with converters, a data compression class, a class for querying hardware capabilities, a dynamic storage class for assembling small objects and primitive types, a registry service and debugging helpers.
- **I/O:** Reading and writing meshes and data to disk is needed by every application. The I/O subsystem of the *gridlib* supports several common 2D and 3D mesh formats. Among them are OpenInventor, NetGen, Alias|Wavefront, OFF, Stanford polygon format, AVS and GSHHS. The I/O subsystem offers a simple interface,

so application specific file formats can be integrated seamlessly, and many *gridlib* developers have done so. For the library user, the subsystem provides additional convenience methods, like file type guessing, textual file type naming and textual representation of the file extension and wildcard, which is very useful for building file-requester dialog boxes.

- **GridGen:** The *gridlib* does not implement grid generation routines apart from simple Delaunay meshing. This subsystem encapsulates an external mesh generator. The resulting mesh is created as a *gridlib* mesh container. The actual grid generator used is therefore transparent to the user. The *gridlib* currently uses GRUMMP [OG02] for grid generation. The second important functionality of this subsystem is to provide methods for transformation of grid representations. Currently, there is a class for creating a progressive representation of tetrahedral grids [LKMG01].

- **Partition:** For parallel and distributed applications, partitioning a 3D grid into pieces in a way such that the number of faces between the partitions is minimized, is an important and non-trivial task. The *gridlib* provides a subsystem that uses the ParMETIS [KSK97] graph partitioning library for computing this property called "least edge-cut". The minimization of the number of shared faces is essential, because it determines the communication volume between partitions of a distributed program.

- **GUI:** This subsystem provides several pieces for controlling *gridlib* functionality with a graphical front-end. Because the usefulness of generic implementations of graphical widgets varies heavily with the application, the subsystem does not try to offer a complete toolbox. Instead, it provides graphical interfaces that are built upon OpenInventor and pure OpenGL to show the developer the general approach to take. For small "viewer-only" applications, complete viewer and control classes are available through simple linking of the subsystem library. The viewers employ the functionality of the visualization and the rendering subsystem to display the grid.

4.6.2 Visualization and Rendering

Many of the distribution and parallelization strategies presented in this thesis have been evaluated while implementing the visualization and the rendering subsystems of the *gridlib*. As with the other client services, they are designed to access the grid through the high-level interfaces and therefore do not interfere with memory layout or other solver-related issues.

The rendering subsystem is split into two major parts: An abstract renderer for geometric primitives and rasterizers. Currently, there are rasterizer implementations for triangles and lines. There is a pure software rasterizer and a wrapper for OpenGL. Both

have several derived children for special rendering contexts or hardware accelerated toolkits.

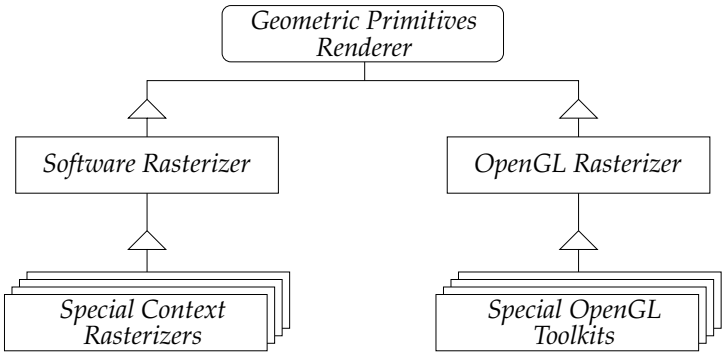


Figure 4.5: The rendering subsystem derives several rasterizers from a well defined interface.

The rendering subsystem is fed by the visualization methods with individual triangles or triangulated meshes. The geometric primitives renderer processes them with standard computer graphics algorithms for culling and clipping in order to reduce the number of triangles to rasterize. Its main task is to serialize the triangles into a specific formatted stream for the rasterizers. This includes computation of vertex attributes like color (probably from a transfer function mapping) and texture coordinates (including loading of texture bitmap). The renderer also features portal culling in order to reduce the overdraw rate of the rasterizers. The geometric primitive renderer is a pure software implementation and can therefore be used for all rasterizers.

For parallel rendering on a supercomputer, the software rasterizer is used [KG01]. It works similar to OpenGL by using a Z-buffer. Although there is some CPU-cycle penalty for the overdraw, this technique allows to implement the scanline code to treat the Z-component similar to other scanline attributes like color and texture. This gives longer code sequences without branches and therefore better optimization possibilities for the compiler. The most appealing aspect of the Z-buffer technique however is, that the rasterizer can be kept quite simple to implement. It finally provides a framebuffer and the Z-buffer to derived children which are left with implementing the transport of the framebuffer content on screen or into an image file. The rendering subsystem provides a distributed version of the framebuffer for NORMA architectures (→ introduction to part one).

G page 145

The visualization subsystem features several algorithms for displaying the simulation results. Again, as the data extraction is done on the mesh level, the algorithms are independent of the actual memory layout. Table 4.1 displays a short summary of the available visualization capabilities (→ color plate B.1). All methods work directly on the unstructured mixed element-type grid and can be applied simultaneously by the viewers.

```
class GeoObject {                 // first some declarations
public:
  virtual void foo() = 0;
}

class Quad : public GeoObject {
public:
  virtual void foo() { ...; }
  void setQuadMarker() { ...; }
}

class Tri : public GeoObject {
public:
  virtual void foo() { ...; }
}

class Vertex {
public:
  virtual void fooSpecial(int i) { ...; }
}

struct MarkQuads {                    // now the two functors
  void operator()(GeoObject *o) {
    Quad *q = dynamic_cast<Quad*>(o);
    if (q) q->setQuadMarker();
  }
  void operator()(Vertex *v) {}
}

struct MakeFoo {
  MakeFoo(int i) : i_(i) {}
  void operator()(GeoObject *o) {
    o->foo();
  }
  void operator()(Vertex *v) {
    v->fooSpecial(i_);
  }
private:
  int i_;
}

Mesh m;                               // now the skeleton program
m.forAllObjects(MarkQuads());
m.forAllObjects(MakeFoo(1));
```

Listing 4.1: The *gridlib* employs the functor concept for algorithmic abstraction on the mesh level.

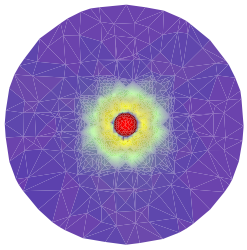
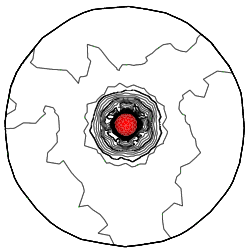
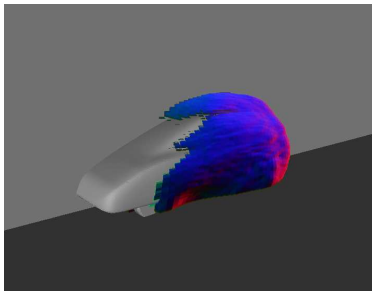
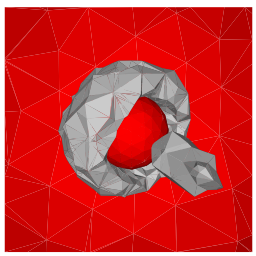
	Planar slices. Scalar values can be mapped onto the slice using a color table. The table can be modified interactively.
	Contour lines on a planar slice. The lines are generated as true geometry and therefore display in correct perspective. The iso-value of a line is annotated.
	Direct volume rendering by regular resampling. The voxel volume can be rendered with hardware support. Interactive color table mapping and gradient shading for visualization of tiny structures and isosurfaces is available.
	Isosurfaces of any scalar value. The surface can be shaded for good spacial impression or another scalar value can be mapped onto it using an interactive color table.

Table 4.1: The *gridlib* offers several visualization algorithms for displaying simulation results.

5

gridlib APPLICATIONS

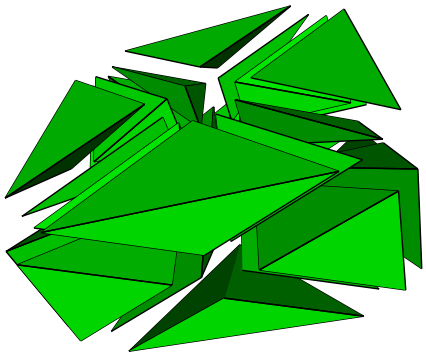
Ce qui est simple, n'est pas vrai,
ce qui ne l'est pas, est inuntilisable.

— *Paul Valéry, Mauvaises pensées*

The *gridlib* has been used for several simulation and visualization applications. Although most of them have not been designed as parallel distributed applications or for the use on a supercomputer, they have triggered and brought forward the development of visualization functionality in *gridlib* considerably [HKRG02, DHH+00a, DHH+00b].

5.1 Evaluating the quality of tetrahedral grids

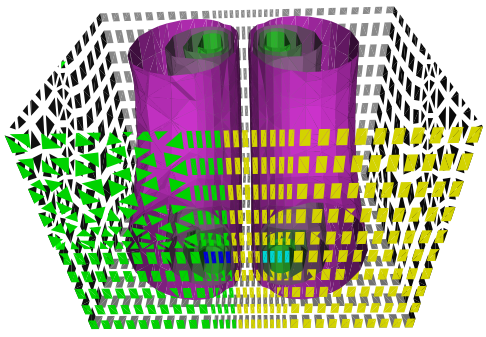
Generating and modifying tetrahedral grids is an important topic for finite element simulations and geometric modeling. Both require well shaped elements. However, the definition of "well shaped" heavily depends on the application. For geometric modeling, the tetrahedra should be "round", i.e. all edges have the same length. For simulation, the elements should be elongated perpendicular to the flow direction for minimal error. In [LKG00], visualization of tetrahedra quality has been implemented (→ color plate B.9). The *gridlib* serves as grid management and algorithmic abstraction library for the quality evaluation routine.



5.2 Progressive isosurfaces from tetrahedral grids

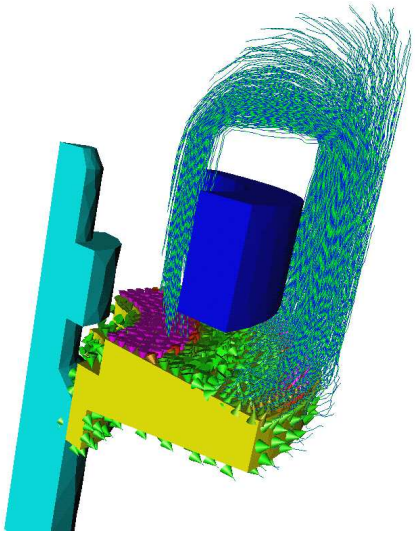
For visualizing large datasets, progressive techniques have proven to be of great value, as the user can interact with a continously refining grid representation while the dataset is still in transmission. This technique has been applied to tetrahedral grids. In [LKMG01], an application has been implemented that also performs isosurface visualization during transmission of the tetrahedral grid. As the grid gets refined, the isosurface is also updated accordingly. This can be done locally, so only updated parts of the surface are exchanged. The *gridlib* is used as a grid management and visualization library (→ color plates B.10 and B.7).

5.3 Fast time-dependent isosurfaces



One of the classic visualization algorithms for scalar data are isosurfaces. For unstructured grids, they are generated with variants of the *marching cubes* algorithm. In [Sch01b], a fast isosurface generation for mixed element-type unstructured grids has been implemented that pre-processes the grid once by sorting the elements according to the scalar values. When traversing the grid, interesting elements that contain the isosurface can be quickly identified by a tree-structure lookup. The *gridlib* has been used as general grid management library and for viewing the result. Using the mesh interface of the *gridlib*, it has been possible to perform the isosurface extraction on a time series of the simulation (→ color plate B.11) to obtain a time-dependent isosurface animation.

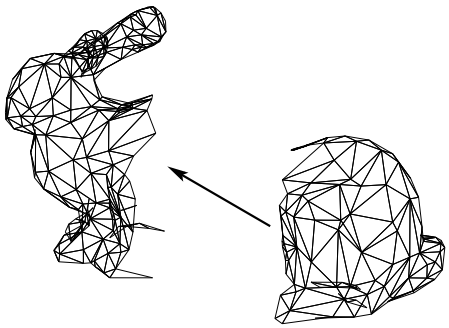
5.4 Visualization across partitions



For studying simulation results, the *gridlib* is used for visualization of multi-domain data. The visualization algorithm is run across partition boundaries (→ color plate B.11). Because the *gridlib* visualization and rendering algorithms are available on the simulation machine, intermediate results can be visualized and the solver can be stopped early if it starts to diverge because of bad parameters or boundary conditions. This can save a lot of CPU time.

5.5 Mesh registration

When assembling larger objects from grid fragments, automatic positioning of mesh pieces is required. Here, the vertices of one piece are registered to the surface of another mesh. The *gridlib* GUI subsystem is used for interactive positioning of the pieces, controlling the optimization process and viewing the result. The grid management and numeric services are used for implementing the registration algorithm.



Part III

Applied Parallelization and Distribution

6

SIMULATION

The reason that data structures and algorithms
can work together seamlessly is . . . that they
do not know anything about each other.

— *Alex Stepanov*

6.1 SIMD processing for Lattice Boltzmann methods

The CFD community has developed several methods for simulating flows. They can be subdivided in finite element approaches and fast fluid simulation methods based on statistical physics that are summarized as Lattice Boltzmann methods [WG00, LL00]. While the former methods compute the flow properties on a possibly unstructured concrete geometry grid, the latter methods use probabilistic approaches on an implicit regular discretization of the domain. In this section, a parallelization strategy using SIMD instructions for Lattice Boltzmann methods is presented.

6.1.1 Lattice gas

The flow properties of a homogenous gas are described by continuum mechanic descriptions from physics research. Numerical simulation of such systems however is very complicated and must take analytic approaches. Apart from algorithmic complexity, good parallel performance cannot be achieved easily, as analytic algorithms usually don't scale well.

For high performance simulation, a good discretization of the domain must be found. From statistical physics, descriptions of gases using discrete particles are known (→ figure 6.1). The particles have different masses, continuous momentum and exhibit random movement and interaction.

The *lattice gas* method constructs a simulation model from these properties by using

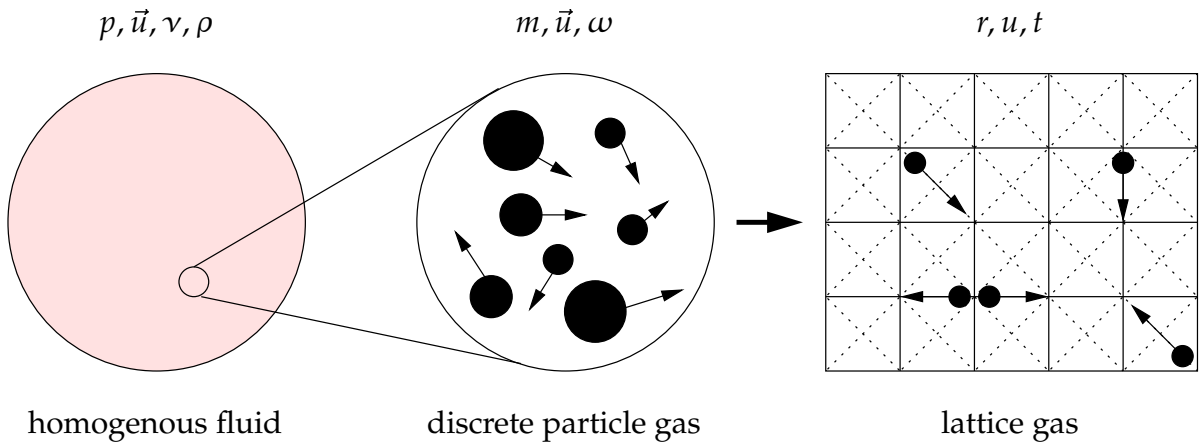


Figure 6.1: From a physical description of a fluid to the lattice gas representation.

unit mass particles with discrete momentum. The movement is realized as an iterative advection using discrete time steps. Because of the lattice structure, particle interaction and collision rules that conserve mass and momentum are simple to implement. The same applies to boundary conditions for discontinuities or walls. The boundary geometry may be complex as it is defined easily by marking the intersected lattice cells to apply a simple "bounce back" rule for particle propagation.

The particles in the lattice can be encoded very efficiently as integer values. The macroscopic values of the continuous gas description can be retrieved from the lattice gas by *ensemble averaging* of the particle distribution: By considering a small local subset of the lattice cells, the density $\rho = \sum \text{particles} / \sum \text{cells}$ and the momentums v_x, v_y are computed by simply counting the respective particle properties. For the particles in figure 6.1, $v_x = (1 + 0 - 1 + 1 - 1)/\rho$ and $v_y = (-1 - 1 + 0 + 0 + 1)/\rho$.

6.1.2 Lattice Boltzmann

The lattice gas method can be implemented very efficiently. Because all calculations are local, parallelization is easy. However, it needs big lattices for good ensemble averaging precision, long averaging times to suppress statistical noise and the flow's parameters are rather difficult to control.

The basic idea of the *Lattice Boltzmann* method is, to circumvent these drawbacks by representing the particle momentum and location as a *particle distribution function* N_i instead of single particles ($\rightarrow$ figure 6.2).

The advection and boundary rules are the same as with the lattice gas method. The particles however need now to be represented by multiple floating point particle densities for each lattice cell. Implementation of particle interaction is feasible by local recalculation of the density distribution (*relaxation*). Therefore, the Lattice Boltzmann method operates on two simple equations. The advection step is given by

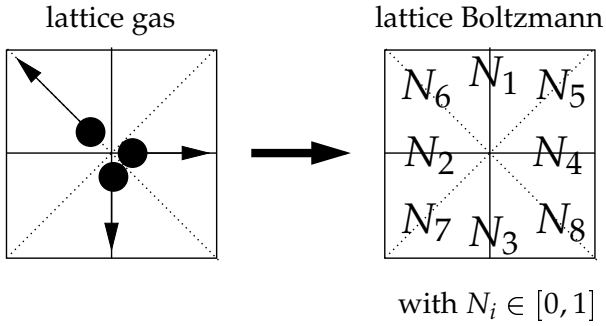


Figure 6.2: From the lattice gas to the Lattice Boltzmann representation.

$N_i(t+1, r+\vec{c}_i) = N_i(t,r) + \Omega(t,r)$ with $\vec{c}_i$ being the connection vector to the neighbor node. The relaxation step is given by $\Omega(t,r) = -\frac{1}{\tau}(N_i(t,r) - N_i^{eq}(t,r))$ using the relaxation parameter $\tau = \frac{6\nu+1}{2}$ and a equilibrium density distribution N_i^{eq}.

The macroscopic quantities fulfilling the Navier-Stokes equations are obtained in terms of the moments of the particle distribution functions.

Density:	$\rho = \sum_i N_i(r,t)$
Flow velocity:	$\vec{u} = \sum_i N_i(r,t)\vec{c}_i/\rho$
Viscosity:	$\nu = \frac{1}{6}(\frac{2}{\tau} - 1)$
Pressure:	$p = \rho c_s^2$

6.1.3 Driven cavity simulation

As the local recalculation of the particle distribution functions N_i is independent for each cell, the Lattice Boltzmann approach is an ideal candidate for SIMD processing, as the relaxation requires no neighbor information. Only the propagation step needs to interchange data. The implementation therefore is structured as in listing 6.1.

```
initDistributions();
while ( timestep && notConverged() ) {
  setBoundaryConditions();
  relaxation();            // <-- SIMD
  propagation();
  timestep--;
}
computeMacroQuantities();
```

Listing 6.1: A Lattice Boltzmann algorithm for driven cavity flow simulation.

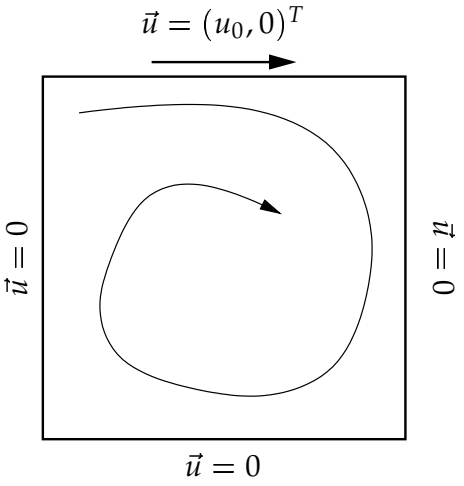


Figure 6.3: Geometry and expected flow within a driven cavity.

The SIMD performance has been evaluated for computing a driven cavity flow experiment. It simulates the flow within a filled cavity. The opening of the cavity meets a very large region that is assumed to have a strong constant flow. The cavity cannot influence the flow within the large region, so the driven cavity model simply asserts a special boundary condition for the boundary cells in question. Figure 6.3 shows the geometry of the experiment and the resulting flow. Depending on the velocity of the driving flow, small currents in the lower corners of the cavity can occur, that rotate counter-clockwise.

To start the simulation, all cells are initialized to $\vec{u} = 0, \rho = 1$ except for the top row, where the driving flow enforces $\vec{u} = (u_0, 0)^T$. The particle density functions are initialized with the simplest equilibrium distribution: $N_i^{eq} = 1$. The advection step uses nine density functions for the possible motion directions in 2D ("d2q9" method). For computing the collision and momentum exchange, the local density and velocity therefore has to be computed as

$$\rho(t,r) = \sum_{i=0}^{8} N_i(t,r) \qquad \vec{u}(t,r) = \frac{1}{\rho(t,r)} \sum_{i=0}^{8} N_i(t,r)\vec{c}_i$$

For efficient SIMD in-cache processing, the cavity is discretized such that each row of cells is stored in a vector class that has an optional 3DNow! implementation of the operators (→ section 3.2). Experiments with cavity edge lengths ranging from 64 to 1024 cells have been performed on three Athlon PC machines with processor speeds of 800, 1200 and 1600 MHz. The fastest system is equipped with a Athlon 1600XP processor which actually runs at 1400 MHz but has additional superscalar hardware. AMD however claims that this processor performs like a conventional processor clocked at 1600 MHz and therefore should be twice as fast as the 800 MHz system. All systems are equipped with identical PC133 SDRAM memory.

G page 145

Figure 6.4 compares the performance of the Lattice Boltzmann simulation in terms

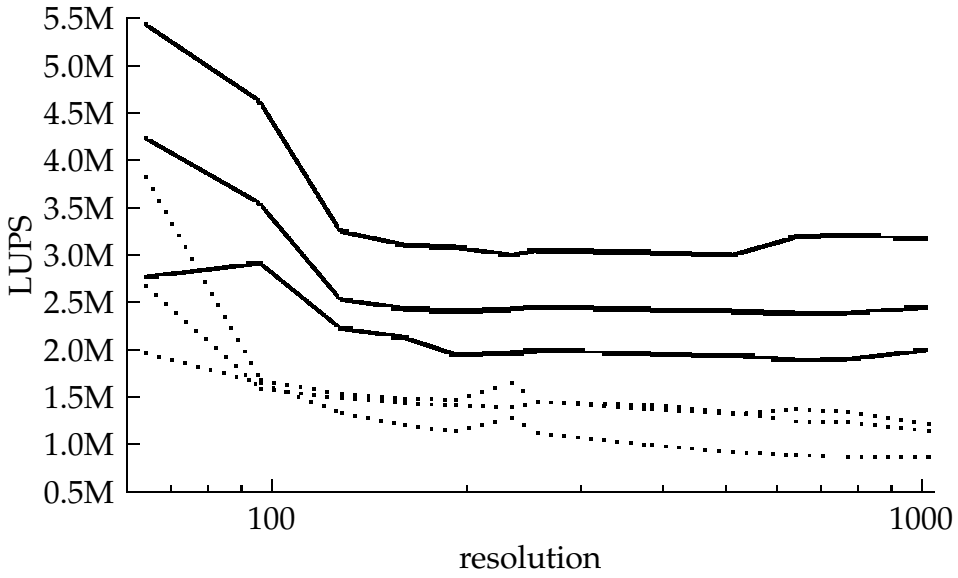


Figure 6.4: Performance of the Lattice Boltzmann solver for Athlon 800, Athlon 1200 and Athlon 1600XP processors. The dotted lines show the performance of the C code, the solid lines show the performance of the SIMD code for each processor.

of *lattice site updates per second (LUPS)* for each processor with 3DNow! SIMD support and without. The SIMD instructions are wrapped within C++ operator implementations, as described in section 3.2. There are several lessons to be learned from these timings:

- Obviously, really good performance is only possible if the vectors fit in the processor cache. This limit is reached already for very small vectors, as the algorithm needs to keep all nine particle distribution functions in the cache.

Facts

- The performance gain for using 3DNow! SIMD instructions represents a factor of three for in-cache vector lengths for all processor speeds.
- The increase in processor speed can not be exploited by the non-SIMD version: The 1600 MHz system offers a peak performance increase of 1.4× compared to the 800 MHz system, even for in-cache vectors.
- The SIMD version is able to maintain a factor of 1.6 increase for very large vectors for the 1600 MHz system compared to the 800 MHz system, although both have the same RAM interface.

The overall conclusion to be drawn from these facts is, that exclusively by using SIMD instructions, the algorithm is capable of using the numerical power of faster processors. Thus, without SIMD, the algorithm is entirely memory-bound and therefore not

scalable. A high performance version for arbitrarily large lattices therefore should employ a tiling strategy and process the lattice in column stripes so the particle distribution function vectors have optimal size to fit in the processor cache (→ figure 6.5).

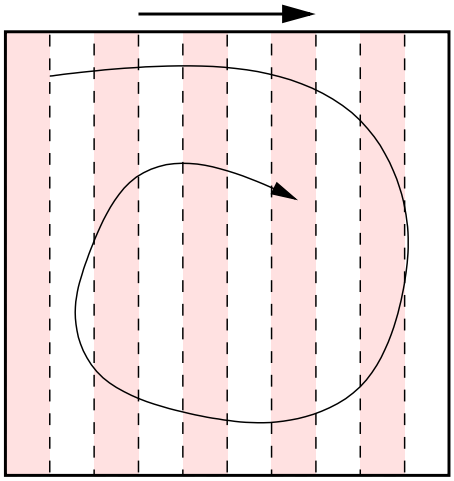


Figure 6.5: Decomposing the domain with vector lengths optimal for SIMD in-cache processing.

As already pointed out in section 3.2, the SIMD instruction code has to be programmed in assembly language. For the Lattice Boltzmann solver, only very simple operations had to be wrapped in C++ operators. Listing 6.2 shows a comparison of the resulting high level program code. The SIMD-enabled code looks a bit more elongated, but conserves its readability and maintainability like the C++ code. This clearly demonstrates, that once the SIMD operations are encapsulated in a C++ class, there are no drawbacks or major influences on higher level programming concepts. Further acceleration of the SIMD code is possible by programming specific operators of the algorithm in assembly language, like the computation of the `square` variable in listing 6.2.

```
// C++
float square = 1.5f*(vx2 + vy2);
float f_eq0 = 4.0f/9.0f*rho*(1.0f - square);

// SIMD
square = vx2;
square += vy2;
square *= 1.5f;
f_eq0.fill(1.0f);
f_eq0 -= square;
f_eq0 *= rho;
f_eq0 *= 4.0f/9.0f;
```

Listing 6.2: Comparing standard C++ code to SIMD-enabled code. Both are equally maintainable.

7

VISUALIZATION

What you see is all you get.

— Brian Kernighan

7.1 Interactive display of time dependent volumes

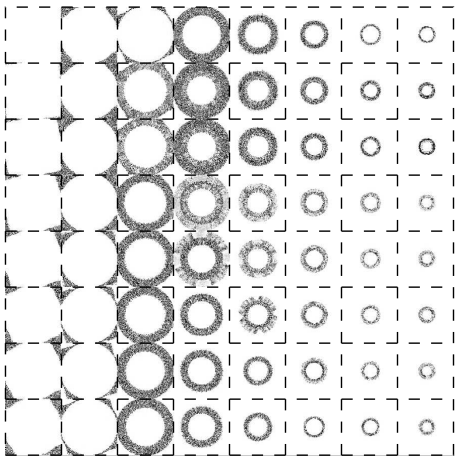
Time dependent simulations produce one result data set per time step. If the results are given as scalar data at discrete positions in space, a voxel volume can be produced by regular resampling. The volume then can be displayed with standard direct volume rendering techniques. If the results are given as vector valued data, the *line integral convolution (LIC)* [CL93] method provides a visualization procedure that gives a good impression of the flow's properties by blurred lines that resemble particle traces. The main idea of the system presented in this section is to decouple the LIC computation and the display of the volume, and to enable integrated handling of stationary and non-stationary flows.

The approach shares the idea of using volume rendering to display the 3D LIC with the work presented by Rezk-Salama et al. [RSHTE99]. In contrast to them, the system enables the integrated analysis of any kind of flow by not restricting the visualization to a single static volume source. It provides a solution to the problem of keeping the volumes of each time-step in memory by using pixel-oriented video streaming techniques. Additionally, it introduces the smooth integration of 3D LIC and particle transport visualization without additional memory or computational costs.

7.1.1 Displaying scalar volumes

In order to efficiently compress and transport the volume voxels, a configurable stream encoder has been implemented. It takes the scalar volumes and slices them in each major direction. The slices get reordered using a well defined enumeration scheme. This produces one stream of slice images for each X, Y and Z. Each stream is compressed

by a video encoder separately. Note that the system therefore implicitly accounts for spacial and temporal coherence of the slices, as supported by the video codec.



G page 145

Figure 7.1: Multiple volume slices are stored in one image to optimize codec usage.

In order to take full advantage of the capabilities of the encoder, it is in general not a good idea to encode every volume slice separately. The stream encoder therefore tries to tile adjacent slices, using a well defined enumeration scheme, into a larger image (→ figure 7.1). The image then gets encoded and is appended to the stream. The encoder takes care to tile the slices in such a way, that there is an equivalent number of images per time-step of the volume in each of the three streams, so it's easy for the player to predict the position of the next simulated time-step within the stream. If the stream player can use 3D textures, only one of the streams is needed. If 3D texturing is not available (or not hardware accelerated), the player can easily search the target stream for the current frame number, if the *shear-warp algorithm* needs to switch the slicing direction.

The main part of the system is the interactive stream player. Its layout is shown in detail in figure 7.2. The pipeline design with ring-buffers between the most CPU consuming components enables efficient multi-threaded asynchronous processing. Additionally, each component can have multiple implementations which take advantage of available hardware support or provide a software solution on other systems.

The main processing steps of the pipeline are (compare figure 7.2 from bottom to top):

- **Reader:** It is responsible for providing the raw stream data. This can be a simple file reader or a subsystem for network access. Its main task is to ensure the continuous data feed to the pipeline. This includes quality-of-service negotiation or caching in the networked case.

- **Stream decoder:** This component interprets the byte stream from the reader by splitting it into content streams. Currently, there is a stream of image data and a stream of configuration commands for the successor components.

- **Image decoder:** The first task is to decode the image data. This includes decompression and re-assembly of the image that has perhaps been split into interlaced chunks or split-fields by the streams format definition. The second task is to perform pixel-oriented image manipulation. This is mostly used for conversion between color-spaces.

➟ **Mapper:** Here, the packing of multiple volume slices into a larger image is reverted. Because this component has knowledge about display properties, this is the place to decide how to extract the slices from the image according to the rendering method (3D textures ↔ shear-warp) to be used. This includes pixel-oriented operations on the slice textures to modify opacity, if the stream does not provide opacity information.

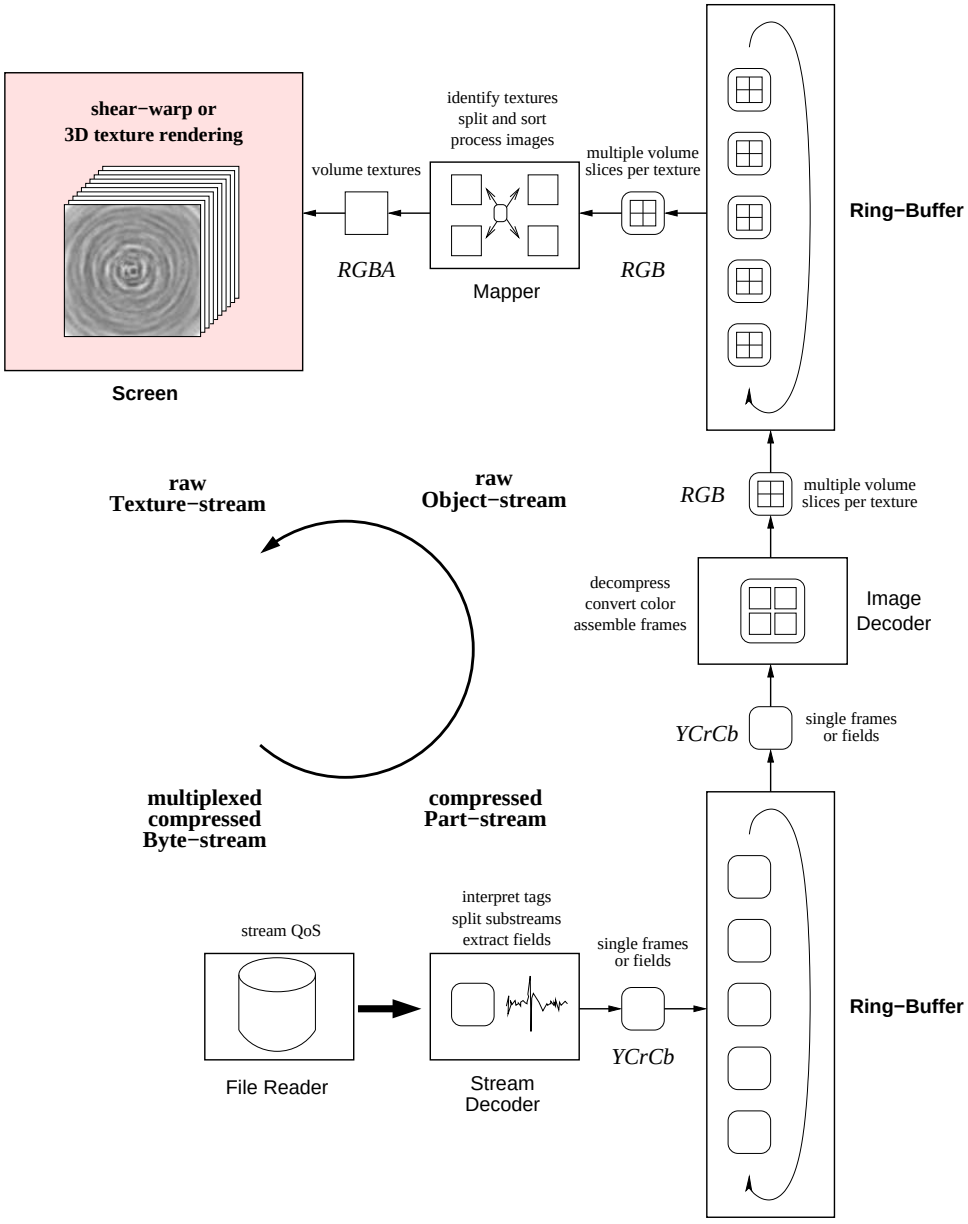


Figure 7.2: The processing pipeline of the interactive stream player (pixel data only).

The pipeline strategy applied here makes the interactive stream player very flexible. As the main caveat with pipeline design is to ensure that the communication bandwidth matches the data production rate of the pipeline steps, special attention has been payed to this: While the reader and the stream decoder are tightly coupled for optimized stream-oriented transport, the two decoders and the mapper are connected by a ring-buffer component. This decouples the writing and reading routines of the connected components and allows input processing at a different granularity than the block size intrinsic to the data. For example, the stream decoder can choose to process the stream as soon as it is available from the reader. This can be useful in case of memory-mapped file access. Alternatively it can block until a certain amount of data has been received and process it in one step in order to avoid frequent context switches when running on a single-processor machine. Note that this strategy also helps the image decoder in case of split-field or interlaced streams, or when the player is connected via a network.

The system described above has been implemented on a SGI O2 workstation. All measurements have been done on a machine with R10000 195 MHz processor and MVP Vice TRE video board. The timings and numbers throughout the remainder of this section refer to a 128^3 3D LIC simulation with 100 time-steps.

The sliced voxel volumes get encoded into Motion-JPEG streams which are written to disk. The encoding of the images is done using the hardware codec of the video board. This produces three shear-warp streams with a size of $\approx$ 18.5 MByte each, which equals a good compression ratio of 1 : 10.8 .

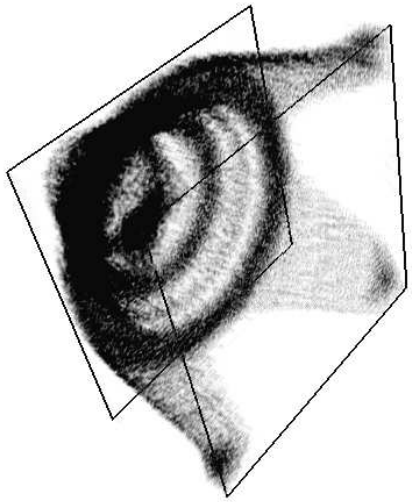


Figure 7.3: Rendering 3D LIC volumes with more than 1024^2 voxels: The border marks the first slice of each stream frame.

The color conversion step in the stream player's pipeline also enables the mapping of additional scalar values onto the volume. Also note that all the possibilities of convenient visual access to interior structures of the volume can also be used: The image decoder and the mapper are using color-table mapping which can easily be extended with the functionality described in their paper without a frame rate penalty. The clipping approach also integrates nicely by introducing a geometry-description-stream to be used for clipping during rendering. This however will degrade the rendering performance dramatically and lead to unacceptable frame rates, like Rezk-Salama et al. have experienced. The color-table-based pixel manipulation therefore clearly is the more efficient way to optimize the visual quality during rendering. The clipping approach however is a very powerful tool during stream creation. The stream encoder can use it to remove unwanted parts of the volume prior to tiling the slices into one stream image.

Because most video codecs have limitations on the image size that can be processed, multi-pass rendering is used if the size of the tiled volume slice image exceeds the codec's capabilities. Figure 7.3 shows a volume with $128^3 = 1024^2 \times 2$ voxels, that has been rendered from two consecutive stream frames with 1024×1024 pixel each. The first slice of each pass is marked by a border. Because each of the frames within the stream is stored in interlaced mode, the player effectively draws 101 time-steps using 202 stream frames composed from 404 fields in this case. This automatically breaks down the block size the codec has to handle to sizes that allow to put the ring-buffers into framebuffer memory. On the test machine, we obtain rates of 9–14 frames per second[1].

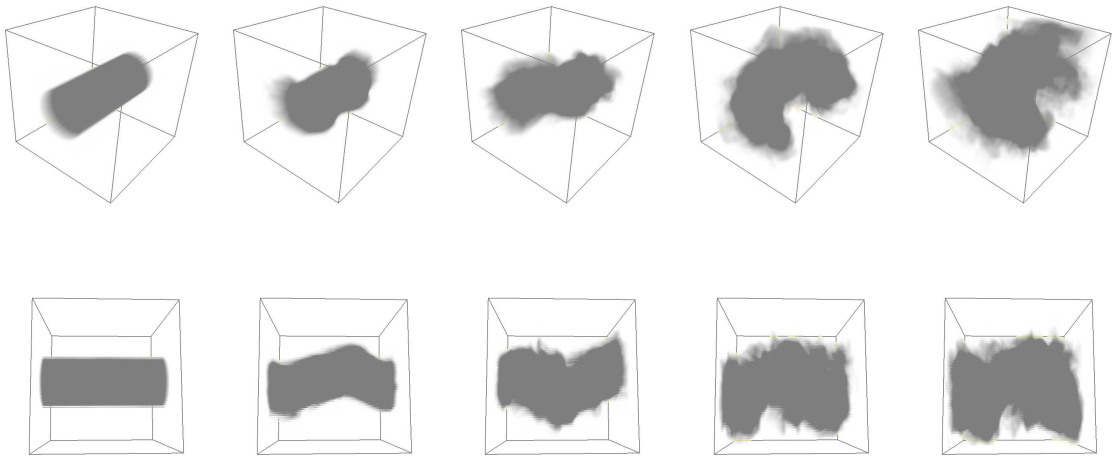


Figure 7.4: Left to right: Two views from different perspectives for five subsequent time-steps of a 128^3 time dependent scalar volume (jet shockwave simulation).

The switching of the texture drawing sequence necessary for the shear-warp rendering technique is supported by the stream decoder component (→ figure 7.2): It has three readers to choose from, each providing a stream for X, Y or Z major rendering direction. Because the seek time for a particular frame in this configuration never exceeds 3 ms when doing jumps at random points in time, the switching of the streams is hardly noticeable to the user. The fast search time is possible, because from the playback, the direction in which to start searching is obvious[2]. Additionally, the latency of the pipeline is entirely hidden by using motion prediction: The stream decoder can be instructed by the mapper to start filling in frames from the target stream to the pipeline in advance to compensate the predicted pre-roll time.

[1]Note that this is independent of the complexity of the voxel volume, compared to clipping [RSHTE99].

[2]In the case of input from a file, the field indices can be cached upon player start[Ben75].

7.1.2 Displaying vector volumes

As depicted in figure 7.5, the system for processing and displaying vector volumes consists of three major building blocks. The first one is a LIC computation on the flow simulation vector field. This is done by a small C++ program wrapping an optimized Fortran LIC kernel. The kernel supports parallel LIC computation on shared memory machines using a static load balancing scheme based on spatial partitioning of the vector field. Using an input volume and a vector field, it creates two output volumes: One, containing the traditional LIC volume and a second one, that contains the input volume distorted according to the vector field.

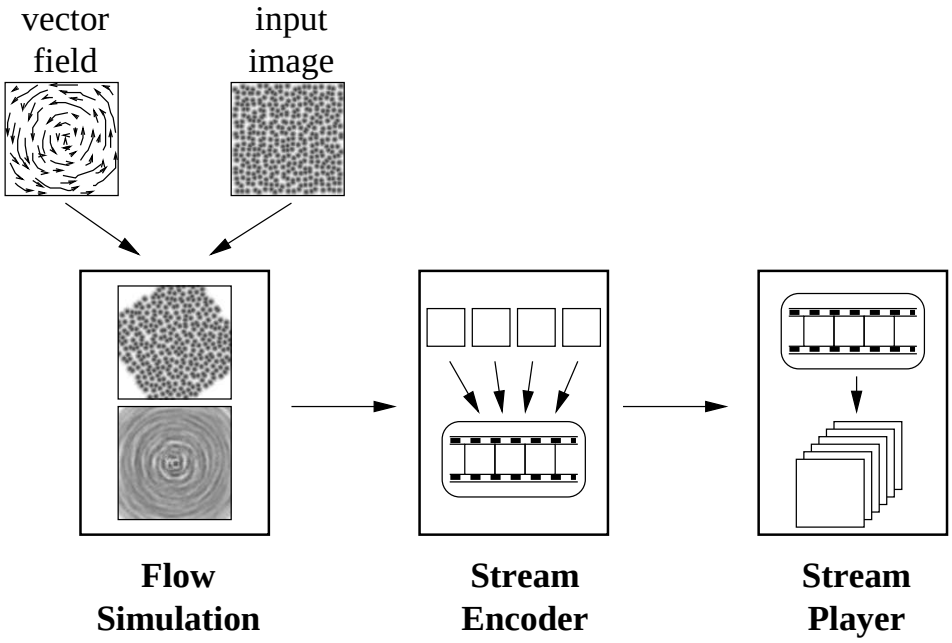


Figure 7.5: The whole system for processing time dependent volumes.

The creation of the second volume is folded directly into the LIC computation within the kernel. It requires no extra computation time, because the LIC algorithm is already sampling the motion vectors of the input field. The warped position of a point in the second volume therefore is exactly the motion vector with maximum length at the original position in the input field. Note that if vector field and voxel volume have the same resolution, this reduces to a simple pixel-copy operation.

The user can choose which volume to process further. In the normal case, this will be the computed LIC volume. In addition to the simulated time-step motion, the blurred LIC lines help the spatial understanding (→ figure 7.6). On the other hand, using the second volume offers a simple way of doing before—after comparisons.

The biggest benefit of creating two volumes is, that the second volume can be used as input volume for the next time-step, enabling pixel-transport visualization. The vector field can be exchanged after each simulation step. Note that this feature enables the

display of non-stationary flow fields. The user simply has to place some non-zero spots in an input field that is zero elsewhere, and the spots will be transported according to the flow.

As the output of the above procedure is a scalar voxel volume, it can be compressed and fed into the stream player as in the previous section. The user first chooses the vector field and defines an initial input flow field configuration. In figure 7.6, the choice was to place a block of 10 slices of white noise in the volume, in order to see how these "particles" get distributed over time. Consequently, the LIC kernel was configured to use the second output volume (the distorted input volume) as input volume for the following time-step. The first output volume (the LIC volume) is sent to the stream encoder, giving some kind of "LIC transport" visualization. Then, the LIC computation of the flow simulation is started on the local machine. Simulating 100 time-steps takes about 22 minutes for the pre-processing step at full resolution. Note that this is in sharp contrast to the work of Rezk-Salama et al. [RSHTE99], where clipping against the time-surface is done during rendering at the expense of frame rate. But from the point of view of a CFD engineer, interactive visualization is the key point, which can be delivered by the presented system.

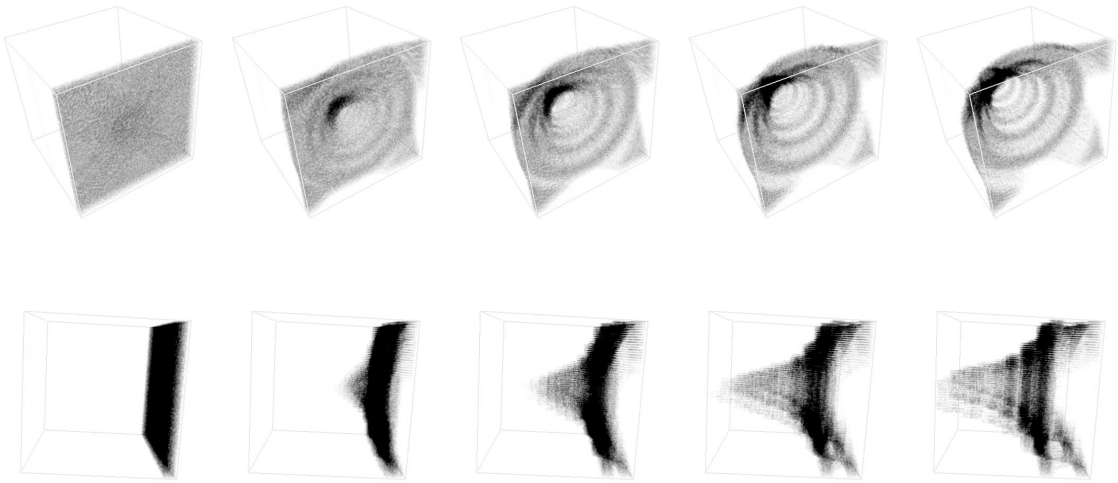


Figure 7.6: Left to right: Two views from different perspectives for five subsequent time-steps of a 128^3 time dependent vector field that has been processed as LIC volume (velocity field with discontinuities).

7.2 Local exact particle tracing

In many scientific areas *computational fluid dynamics (CFD)* simulations are of central importance. The result usually describes the flow behavior in a 3D environment. To

understand the characteristics of the simulated process, a meaningful visualization of the data is necessary. The raw data is defined on a discrete structure, a grid. At the nodes or vertices of the grid the velocity (a vector field) and other simulation data, e.g. pressure, density or energy (scalar values) is stored.

One of the most common methods of acquiring knowledge of flows is the use of particle tracing [USM96, NJS+97, Frü94]. It is also the basis of many other visualization techniques such as streak-lines, streak-ribbons, time-surfaces or line integral convolution (→ section 7.1). The particle tracing visualization method shows the trajectory of one mass-less particle in the flow. The trajectory is obtained by the integration of the ordinary differential equation corresponding to the vector field.

The integration is usually done numerically. In [KRG02], another approach for vector fields defined on a tetrahedral mesh has been implemented within the *gridlib* project. It is a modification of the *exact integration method* introduced by Nielson and Jung in [NJ99]. This method traverses every cell in a single step. The modification comprises a sophisticated parallel pre-processing of the data, which results in a classification of the cells and provides for each cell sufficient information to perform the exact integration very fast. The method determines the exact exit points of the particle for every cell that is traversed. The particle trace is the polyline of the computed points or an interpolating curve. In contrast to all the numerical integration schemes, this approach guarantees that the local errors will not accumulate. Hence it is globally very accurate compared to methods built on numeric integration. See [LB98] for a comparison of the achievable accuracy for numeric integration schemes.

As the starting point of the particle trace is selected by the user, the method must be fast enough to be usable in interactive applications. At the same time it may not produce incorrect results. Once a characteristic streamline is found, the user may wish to display it most faithfully to the physical environment. Here, more expensive rendering methods are acceptable. The streamline however must stay the same, so switching numeric methods is prohibitive as it may result in particle traces shaped differently. We present a visualization strategy that offers interactive rendering as well as high-quality ray tracing output based on the same particle path building routine.

7.2.1 Integration methods

The classic Euler and Runge-Kutta integration methods mainly differ in the provided accuracy related to the numerical integration itself. They do not take geometry issues into account. Therefore, for achieving the same accuracy as the local exact method, the classic integration methods would need much more and smaller steps. The Runge-Kutta $\mathbf{RK}_2$ method for example is less expensive than the local exact method, but the number of required steps for obtaining the same accuracy outweighs this advantage by far.

Table 7.1 shows the overall time needed by distinct integration methods to compute streamlines for 1000 integration steps. The calculated paths consist of an identical number of segments with the same overall length, so the average step size is the same.

The paths can be different, because the methods offer different accuracy. In order to achieve the same numerical accuracy as the local exact method, the Runge-Kutta methods would need smaller step sizes, resulting in longer processing times. As the last two lines of the table demonstrate, the local exact method is very competitive if the pre-processing suggested in [KRG02] has been performed. The table has been generated for the Sphere dataset, for which the pre-processing took 5.4 seconds. If the processing of the local exact method is delayed and performed on-the-fly when the trajectory enters a cell, no additional memory is needed. This demonstrates the ability of the algorithm to trade memory for speed. All time measurements are wall-clock time on a SGI Octane 300MHz R12000.

integration method	time (sec)
Euler	1.75
Heun	2.31
Runge-Kutta $\mathbf{RK}_3$	3.11
Runge-Kutta $\mathbf{RK}_4$	3.33
Local exact with pre-processing	2.63
Local exact on-the-fly	9.22

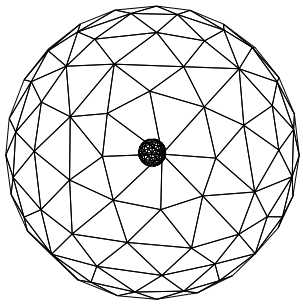


Table 7.1: Timings for different integration methods for 1000 integration steps on the Sphere dataset.

The advantages of the local exact method can be briefly summarized:

PRO / CON

✓ The local exact method gives the exact solution, provided that the numerical errors are neglected, and one assumes that the underlying linear interpolation of the vector field is a valid method.

✓ For the local exact method no point location is necessary. In fact, knowing the neighbors of a tetrahedron, the next tetrahedron to be processed is given by the location of the exit point.

✓ The local exact method traverses each cell in a single step. Therefore, the step size need not to be specified, nor does one have to take care of (adaptively) modifying it. The adaptation is built-in: For simulation, typically in numerically critical areas or areas of special importance, one has a fine mesh of tetrahedra. In these areas, the path segments produced by the local exact method are then small as well.

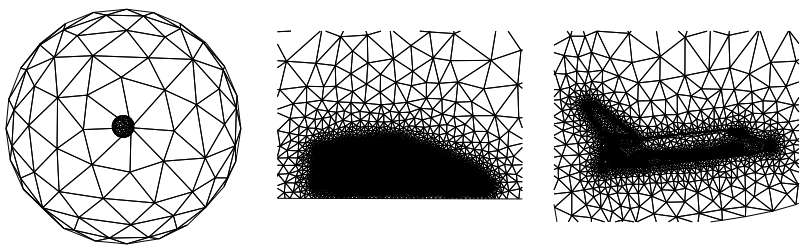
✗ The local exact method does need some additional memory. However the memory requirement can be scaled from storing all pre-processing data, to computing the pre-processing data on-the-fly for each tetrahedron visited. The data may be cached for reuse or discarded immediately.

7.2.2 Cell classification

When the path line enters a cell, the intersection point of the exact solution and the exit face of the tetrahedron is determined. The exit point is the entry point to the following cell, where the method is repeated. The final particle path is the concatenation of the intersection points. In contrast to [NJ99], the method used in *gridlib* does not pay attention to all possible special cases. The tetrahedra are classified as follows:

- **Parallel cells** are cells whose vector field does not change the direction, i.e. all four velocity vectors point in the same direction. Typically ten percent of all cells belong to this group.
- **Normal cells** have three different eigenvalues as the linear part of the (linearly interpolated) vector field. In addition, the critical point of the vector field is located outside the tetrahedron.
- **Extraordinary cells** are all the remaining ones. Typically less than 0.5% of all cells are of this type. Table 7.2 compares three randomly selected CFD data sets.

In the case of an extraordinary cell, a Runge-Kutta integration scheme is used until the cell search routine detects a new cell. As table 7.2 shows, this case occurs very seldom. Therefore, only extraordinary cells need to be traversed using multiple steps. The vast majority of cells is traversed in a single step using highly accurate integration.



# of cells	Sphere	Car	Shuttle
total	8,193	457,874	1,058,785
extraord.	15	1,247	3,232
preproc. time	5.4 sec	112.8 sec	219.2 sec

Table 7.2: Number of extraordinary cells in local exact particle tracing.

7.2.3 Parallel pre-processing

The pre-processing step for the local exact method consists of computing the eigenvalues and eigenvectors of the tetrahedron that build its eigenspace. For doing the eigenvalue analysis, the linear interpolation $\vec{\mathbf{v}}(\vec{x}) = A\vec{x} + \vec{o}$ of the vector field must be determined. The 3×3 matrix A and the translation vector $\vec{o}$ are uniquely determined by

the four vector equations $AV_i + \vec{o} = \vec{v}_i$, $(i = 0, 1, 2, 3)$, where V_i are the vertices of the tetrahedron and $\vec{v}_i$ are the attached velocities.

```
#pragma omp parallel for private(cell,A,lambda,crit)
for (int i=0; i<grid.numCells(); i++) {

  Cell cell = grid[i];

  // check velocity field
  if (cell.getVelocities().computeDeviation() < EPSILON)
    cell.mark(PARALLEL);
  else {
    Matrix A = LinSolve(cell.getVertices(), cell.getVelocities());
    Real lambda[3] = A.getEigenvalues();

    // check for singularities
    if (lambda.same)
      cell.mark(EXTRAORDINARY);
    else if (lambda.real)
      cell.mark(REAL);
    else
      cell.mark(COMPLEX);

    cell.setEigenvectors(A.getEigenvectors());

    // check location of critical point
    Vector crit = (-A).inverse() * o;
    if (crit.isInside(cell))
      cell.mark(EXTRAORDINARY);
  }
}
```

Listing 7.1: The eigenvalues and eigenvectors of all tetrahedra are computed in a parallel pre-processing step.

The eigenvalues of A must be different from each other. This condition is asserted by classifying the tetrahedron by first checking the velocity vectors $\vec{v}_0$, $\vec{v}_1$, $\vec{v}_2$ and $\vec{v}_3$ at the vertices. If they all point in the same direction (with respect to some threshold), the cell gets labeled *parallel* and the direction vector is stored. Parallel cells are treated differently when building the particle path line.

For non-parallel cells, a complete eigenvalue and eigenvector analysis of A is performed. The eigenvalues are computed using Cardano's formula, which allows the analytical calculation of the roots of the polynomial (which is of order three in this case). Now the tetrahedron is labeled *real* if all its eigenvalues are real. It is labeled *complex* if two of them are complex conjugates.

Finally, the tetrahedron is labeled *extraordinary*, if the eigenvalues are the same (with respect to some threshold) or if the critical point $\vec{x}_{crit}$ of the linearly interpolated vector field lies within the tetrahedron:

$$A\vec{x}_{crit} + \vec{o} = 0 \quad \rightarrow \quad \vec{x}_{crit} = -A^{-1}\vec{o}$$

Listing 7.1 shows the pseudocode for the algorithm. Note that the calculations for each cell are independent, which allows to employ the OpenMP parallelization strategy.

7.2.4 Building a smooth curve

In order to make the streamlines look more pleasant, the segment within one grid element should be represented by a cubic curve. From the pre-processing step, the exact exit point $\vec{x}(t_{out})$ on the tetrahedron and the exit time t_{out} is known. The entry point of the tetrahedron $\vec{x}(t_{in})$ is the exit point of the previously processed cell. By defining the cell time $t_{cell} = t_{out} - t_{in}$, the corresponding velocity vectors $\vec{v}_{in}$ and $\vec{v}_{out}$ are obtained by linear interpolation. Now the unique Bézier curve interpolating position and velocity can be calculated easily using Bernstein polynomials.

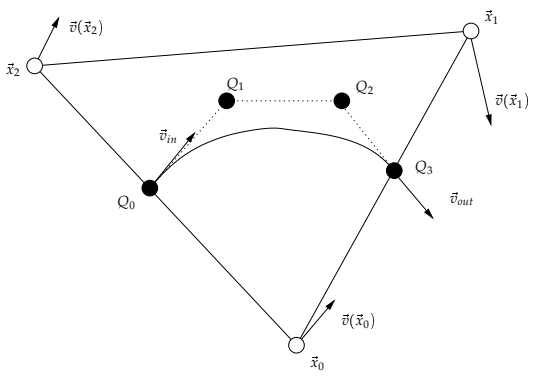
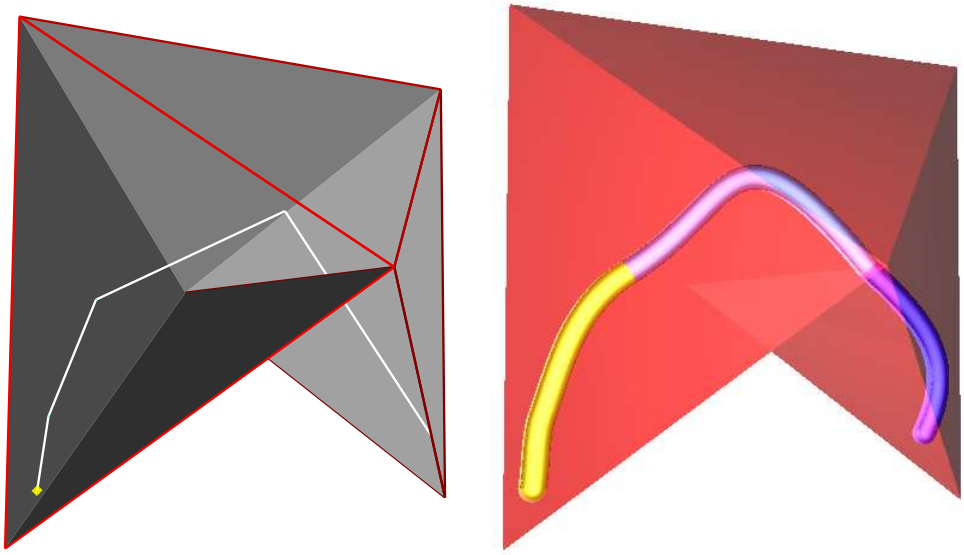


Figure 7.7: Building a simple Bézier spline for a smooth particle curve.

In addition, please note that the the Bézier spline (→ figure 7.7) being the Hermite interpolant, has the better local approximation error $O(h^4)$ compared to $O(h^2)$ for the linear interpolant.

The local exact particle path faithfully follows the true vector field. The figure 7.8(a) shows the trajectory computed in a synthetic three-tetrahedra dataset displayed using OpenGL line primitives. The flow vectors of the outermost vertices are perpendicular and point up, right and down when going from the left tetrahedron to the right one. Traditional integration methods need a small step size to produce an equivalent line. The local exact algorithm computes the particle trace in just three steps. In the ray traced image 7.8(b), the three segments of the particle curve are emphasized by different colors. Figure 7.9 shows particle trajectories in the simulation data set “car”, see also color plate B.8.



(a) (b)

Figure 7.8: High-quality visualization using the local exact integration method for particle tracing. In (a), the trajectory is rendered with OpenGL. The smooth ray traced curve in (b) is colored differently in every segment computed. See also color plate B.6

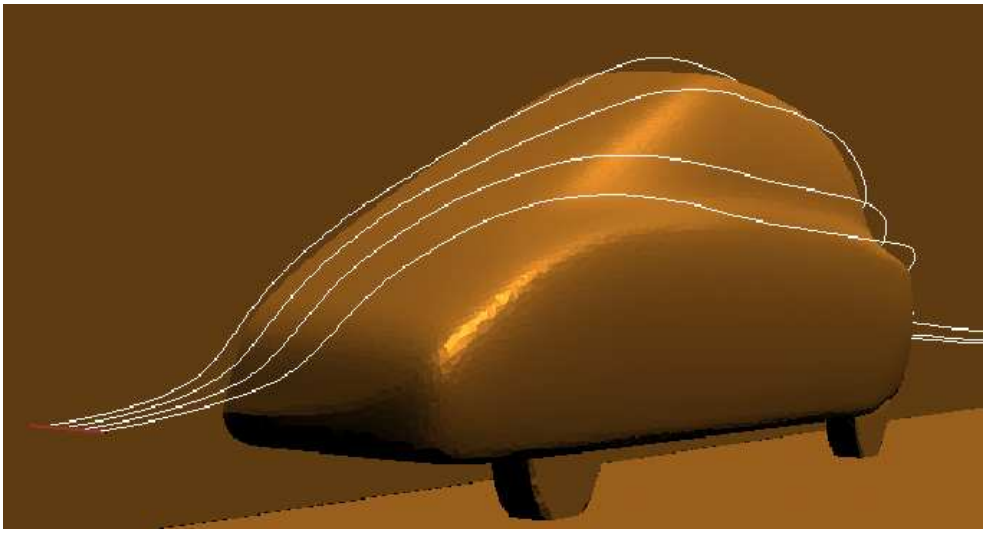


Figure 7.9: Car model in the wind tunnel computed with the local exact method.

8

RENDERING

Software is hard.

— Donald E. Knuth

8.1 Transparent Distributed Processing For Rendering

Rendering, in particular the simulation of global illumination, uses computationally very demanding algorithms. As a consequence many researchers have looked into speeding up the computation by distributing it over a number of computational units. However, in almost all cases the relevant algorithms have been completely redesigned in order to achieve high efficiency for the particular distributed or parallel environment. At the same time, global illumination algorithms have gotten more and more sophisticated and complex. Often several basic algorithms are combined in multi-pass arrangements to achieve the desired lighting effects. As a result, it is becoming increasingly difficult to analyze and adapt the algorithms for optimal parallel execution at the lower levels. Furthermore, these bottom-up approaches destroy the basic design of an algorithm by polluting it with distribution logic and thus easily make it unmaintainable.

In this section, a top-down approach for designing distributed applications based on their existing object-oriented decomposition is presented [KS99]. Distribution logic given through the coarse grained CORBA parallelization and distribution strategy, is introduced transparently to the existing application logic. The design approach is demonstrated using several examples of multi-pass global illumination computation and ray tracing. The results show that a good speedup can usually be obtained even with minimal intervention into existing applications.

There is a large number of papers on parallelization and distribution of rendering and lighting simulation algorithms. Good surveys are available in [RCJ98, CR98, Cro98]. Most of the papers concentrate on low-level distribution for achieving high performance. One of the few exceptions is the paper by Heirich and Arvo [HA97] describ-

ing an object-oriented approach based on the Actor model. Although their system provides for location and communication transparency, the distribution infrastructure is still highly visible to the programmer.

8.1.1 Distributed Lighting Networks

The presented distribution framework has been developed for the Lighting Network subsystem in the VISION rendering framework [Slu96, SSS98, SSH+98]. Individual Renderer and LightOp objects are to be distributed across a network or to be run in parallel through the use of threads. Color plate B.5 shows an example network that computes high quality lighting using both direct and indirect illumination algorithms.

For the implementation, the basic operating system functions are accessed via the operating system adaption layer interface of the ACE library (→ section 1.3.1). The communication and remote object creation is done using CORBA (→ section 1.3.2). To facilitate further development and maintenance, the design of the base classes follows the guidelines of several design patterns [GHJV95, CS98, LS96, SHP97, McK95].

In the following, an integrated approach to parallelization and distribution of application modules is presented. It is based on the fact, that object-oriented systems should be and usually are composed of several quite independent subsystems. In contrast to addressing parallelization at the level of individual objects, larger subsystems of objects usually offer a better suited granularity for distributing the work across computers. These subsystems are often accessed through the interface of a single object using the "facade" design pattern. In an application based on this common design approach, these few facade classes can easily be mapped to CORBA interfaces [OMG97b], providing the basis for distributing the application. However, this initial step does not solve the problem completely, as the CORBA-specific code would be introduced at the heart of the application and the details of distribution should not be visible to a developer.

G page 145

8.1.2 Design Patterns for Transparent Distribution

In order to make the distribution and parallelization related parts transparent to the programmer of rendering or lighting algorithms, a new distribution interface that completely hides the CORBA distribution infrastructure from the application is built. The new interface provides the illusion of traditional, non-distributed classes to the outside, while internally implementing optimized distributed object invocations. It is based on asynchronous communication with a multi-threaded *request-callback scheme* [SV96] to enable a maximum of parallelism. Additionally, the framework performs load balancing and bundling of requests to avoid network latencies. For encapsulating existing interfaces, the framework provides base classes that carry management services for object creation, communication transport control and synchronization and many other services (see below). The wrapper for the subsystems that contain the rendering and illumination algorithms use and inherit from these base classes.

Wrapping for Distribution

In order to actually reuse the existing object implementations within a distributed environment, the distribution framework provides wrappers for entire subsystems. A wrapper actually consists of two half-wrappers that encapsulate the subsystem as a CORBA client (calling) and as a server (called) (→ figure 8.1(a)). Assuming that a subsystem is represented by at least one abstract C++ facade class and communicates with the outside through interfaces defined by similar facade classes, replicating each of these interfaces in CORBA IDL [OMG95a] gives a complete distributable system for the existing implementations. Additionally, new methods that allow for the bundling of multiple requests on the calling side are defined. The server side is implemented by forwarding the requests to the wrapped facade object in a pseudo-polymorphic way (→ section 4.4.1), serializing any bundled messages that arrive, and managing asynchronous calls (→ figure 8.1(b)).

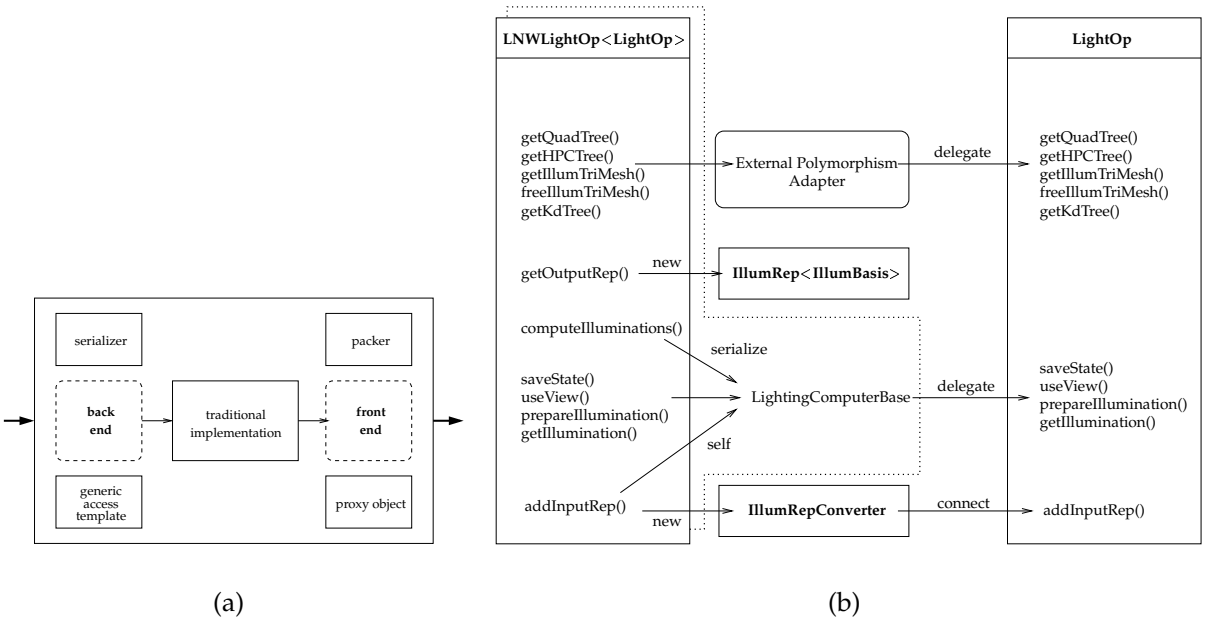


Figure 8.1: Reusing existing implementations without code modifications through wrappers.

For the client role of a wrapped subsystem, C++ classes need to be instantiated that derive from a distributed C++ proxy template. They translate the calls from the original C++ interface to calls, that use the CORBA object references. This layer is also responsible for bundling individual calls and using the new asynchronous interface methods for bundled requests within the CORBA interface.

Although this wrapping seems complicated and does require some small amount of manual coding, most of the work can be delegated to generalized template abstract base

G page 145

classes. When viewed from the outside, the encapsulated subsystem looks just like a distributed CORBA object using the equivalent CORBA IDL interface. To the contained object, the wrapper looks exactly like any other part of the traditional system using the original C++ interfaces.

The biggest benefit of using this kind of wrappers is the possibility of reusing existing code. While this does not take advantage of parallelization within a subsystem, it enables the distribution and parallelization of different subsystems. This can be of great value, in particular when multiple memory-intensive algorithms have to be separated across multiple machines. The interfaces, provided by the wrappers, finally allow wrapped traditional objects to transparently cooperate with other distributed objects.

Replication and Request-Multiplexing

In order for old code to use distributed subsystems, an additional wrapper is necessary. Its interface is derived from the original C++ facade interface, but it translates the messages to corresponding calls to distributed CORBA objects, e.g. those from the previous section.

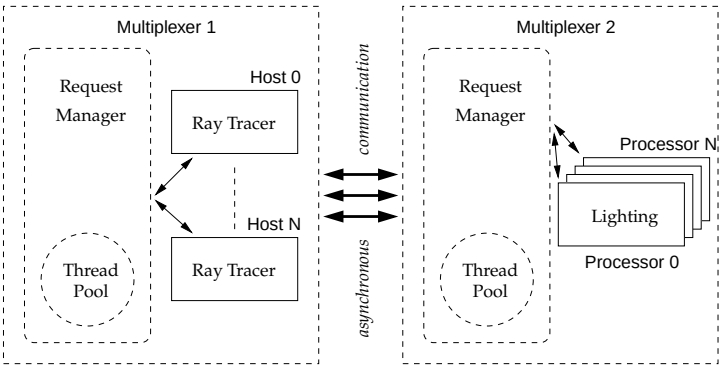


Figure 8.2: A multiplexer can distribute or parallelize computation objects through replication.

As mentioned before, this translation has several aspects. For one, it translates between traditional and CORBA types where object data needs to be copied into IDL structures. Second, small individual requests may be accumulated and sent across the network in bundles, thus avoiding network traffic overhead. In addition, we take the opportunity of the wrapper to perform multiplexing and re-packeting of requests across a pool of functionally identical CORBA servers. This allows to distribute the computational load evenly using load balancing performed by the wrapper. However, because of the current synchronous nature of CORBA method calls, multiplexing needs to use the request-callback scheme provided by the base classes of the framework.

Load balancing is performed by sending requests to the server with the lowest load. To this end, the servers maintain FIFOs of requests to balance network latencies. The fill-level of those FIFOs is communicated back to the wrappers piggy-packed on data returned in the callbacks. Using this scheme, the multiplexed classes look to the outside like a single, more powerful instance of the same subsystem (→ figure 8.2). The benefit of this approach is that by using wrappers and multiplexers, existing code can fairly easily be wrapped, replicated, and thereby sped up. While multiplexers fan out requests, the wrappers introduced in the previous section automatically combine and

concentrate asynchronous requests from multiple clients. Note that both patterns perfectly meet the goal of distribution transparency and do not alter the application logic of the remaining system at all.

The pseudocode in listing 8.1 shows how a multiplexer for lighting computations inherits the interface of the lighting base class and overloads the computation request method by implementing some scheduling strategy (→ figure 8.1(b)).

```
IDL:

interface LightOp {
  void computeIlluminations(in sequence<Request> req);
};

interface Multiplexer : LightOp {
  void addLightOp(in LightOp op);
};

C++:

class Multiplexer : public IDLmultiplexerInterface {
  virtual void addLightOp(LightOp op) {
    lightOpList_.push_back(op);
  }
  virtual void computeIlluminations(Request req[]) {
    int idx= determineBestServer();
    lightOpList_[idx]->computeIlluminations(req);
  }

  protected:
  vector<LightOp> lightOpList_;
};
```

Listing 8.1: A multiplexer for lighting computations is build by inheriting the lighting interface and overloading the request method.

The performance of the asynchronous communication pattern of the multiplexers and wrappers is shown in table 8.1. It compares the packeted data transfer within a small lighting network using asynchronous requests with an equivalent network using the original interface with fine granularity. Both cases use wrapped traditional LightOps and the same host configuration:

SGI	Onyx	Onyx	O2
# processors	4	2	1
R10k @ MHz	196	195	195
Renderer			×
Lighting	Irr. Grad.	Direct	Combine

wallclock seconds for	**asynchronous LightOps**	**wrapped-only LightOps**
Session Setup	22.26	23.37
Parsing Scene	5.80	5.67
Lighting Setup	1.56	1.68
Renderer Setup	0.30	0.34
Init Operators	25.38	26.91
Render Frame	1,922.06	2,916.95
Total	**1,977.36**	**2,974.92**
	66 %	100 %

Table 8.1: Packeted asynchronous data transfer within a lighting network compared to LightOps using CORBA's synchronous request invocation.

Transparent Services

Some subsystems are computational bottlenecks and promise to offer substantial speed-up when they are completely re-implemented to take advantage of distribution.

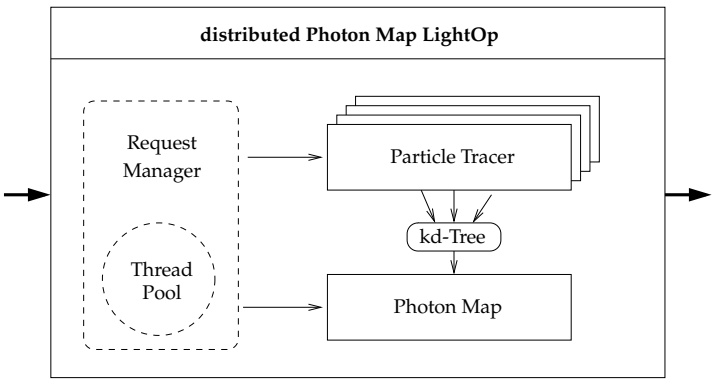


Figure 8.3: The framework offers several services for writing distributed computation classes.

The framework provides distribution and parallelization services within the wrapper classes that go beyond plain data transportation and interface adaption, such as thread-pool handling, mutexes, factories for one-to-many and many-to-one operating threads and their synchronization, runtime system state and type information.

This pattern is the most powerful form of creating a new computational object for the distributed system. It does however require knowledge about the design and behavior of the distribution services. Because the wrapper classes provide the CORBA interface to the other traditional subsystems of the framework, a distributed or parallel implementation of a subsystem can easily access them directly. A good example is a class

that performs distributed lighting computation using the PhotonMap algorithm [Jen96] (→ figure 8.3). Existing code for tracing of photons from the light sources and for reconstructing illumination information can easily be reused. Both reused object implementations are wrapped with the patterns described above. Because the algorithm is aware of its distributed or parallel nature, it can steer and adapt to the computational requirements, e.g. by adding new particle tracer threads on a multi-processor machine or adding new instances of distributed objects. This scheme allows the programmer to gradually make selected subsystems aware of the distribution infrastructure without compromising the remaining system on the way.

8.1.3 Discussion

The programming design patterns introduced above offer several benefits:

Facts

- New developments within the traditional framework are immediately distributable through the wrapper pattern, which offers speedup through replication and multiplexing.
- There is no need for developers of algorithms to bother with distribution and parallelization issues because the distribution framework does not alter or interfere with the application logic.
- The distribution and parallelization services offered by the framework provide the developer of advanced computation classes with basic functionality that is guaranteed to conform to the overall design.
- The learning effort for beginners can be reduced dramatically by a transparent distribution infrastructure – in particular if compared to other distribution frameworks and the large number of new software concepts introduced by them.
- The distribution framework transparently supports modularization and helps to structure the framework into toolkits with well defined interfaces. This can help to reduce the overall programming effort, and promotes a better understanding of the big picture.

For the VISION rendering system, distributed LightOps have been implemented using the design patterns from above. In order to reuse the traditional LightOp implementations efficiently, several multiplexer classes are available along with different scheduling strategies. This allows for building distributed lighting networks, that functionally distribute lighting calculations. The configuration of the distributed objects is usually specified in a TCL configuration file using the existing scripting engine of the traditional VISION system, avoiding the introduction of a second tier of abstraction for configuring the distributed system (unlike [Mer99]).

Distributed Rendering

To optimize rendering times in the case of calculating previews or testing new computation class implementations, the following configuration of a distributed VISION system shows the best achievable speedup found when using the framework. It uses 4 hosts with a total of 8 processors. There are 8 ray tracers to work in data-parallel mode and 6 lighting modules. Each group is controlled by a multiplexer. The distribution framework ensures that all communication between the two multiplexers is done asynchronously. Here is the setup:

SGI	Onyx	Onyx	O2	O2
# processors	4	2	1	1
R10k @ MHz	196	195	195	195
Renderer	4	2	1	1
Lighting	4	2	-	-

The lighting hosts execute a traditional implementation of an Irradiance Gradients [WH92] LightOp, which is wrapped for distribution. Additionally, the wrappers on the multi-processor machines also include a multiplexer that executes the incoming requests in parallel using a thread pool. Because there are multiple threads per CPU, the multiplexer synchronizes them in order not to overload the machine.

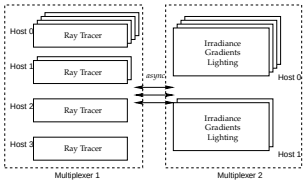
Table 8.2 compares the distributed system to the traditional VISION system with a single thread of control, running on the fastest machine in a single address space and calculating lighting with the very same LightOp implementation. The speedup obtained is near the theoretical maximum of 12.5%. The overhead of ≈90 seconds consists of 30 seconds session setup, 5 seconds of additional parsing on the CORBA startup client and another 5 seconds delay for allowing the hosts to clean up the CORBA objects before the main CORBA startup client shuts down all VISION instances. After subtracting this overhead, a penalty of ≈13% remains during the rendering phase for the distributed system. This is a very good result, given such a general and unintrusive distribution infrastructure.

Distributing Complex Lighting Computations

The functional decomposition of a lighting network offers the biggest potential for distribution and parallelization, at the risk of high communication costs. The asynchronous request-callback communication paradigm is able to provide a partial solution for that problem. The following configuration uses 3 hosts with a total of 7 processors:

SGI	@ MHz	# proc. R10k	Renderer	Lighting
Onyx	196	4		Photon Map, Direct, Combine
Onyx	195	2	×	Photon Map, Irrad. Grad.
Octane	175	1	×	Photon Map

wallclock seconds for	**distributed System**	**traditional VISION**
Session Setup	31.91	-
Parsing Scene	5.61	-
Lighting Setup	0.14	-
Renderer Setup	0.36	-
Init Operators	32.36	20.95
Render Frame	317.03	2,359.20
Total	**387.41**	**2,380.15**
	16 %	100 %



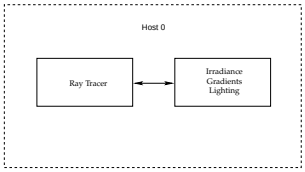


Table 8.2: A distributed system using two multiplexers, controlling the data-parallel renderers and the lighting objects on the left side, are compared to the traditional single-threaded system.

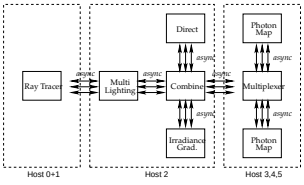
In this setup, the reconstruction method of the PhotonMap LightOp takes much more time to process a request, than any of the other LightOps in the lighting network. Consequently, a multiplexer is used to distribute this LightOp onto 3 hosts. In contrast, the three other LightOps are executed on multi-processor machines, because their reconstruction method is fast and the communication between them can be optimized, if the CORBA implementation supports object collocation. In order to drive this complex lighting subsystem, two hosts execute rendering objects controlled by a multiplexer in a data-parallel way.

As table 8.3 shows, the speedup obtained by this setup is not as good as in the previous example. But even the advantage of the non-distributed version of running in a single address space, does not outweigh the communication overhead of the distributed system. Profiling shows, that the performance difference to the theoretical maximum of 14.3% is mainly due to process idle times. This occurs for example, if the calculation of one upstream LightOp is sufficiently delayed. Since the underlying Lighting Network is entirely pull-driven, the pipeline is blocked. The framework therefore allows the asynchronous interface to drive three parallel streams at a time. Additionally, the resource handling within the base classes allows running the rendering computation concurrently with a lighting computation, resulting in a kind of interleaving CPU-usage scheme, if the lighting pipeline on the host is stall.

This example shows that there are cases where the full transparency of the distribution infrastructure cannot hide inherent limitations due to coarse grained communication patterns of existing subsystems. Note however, that this behavior is mostly a problem of the non-distribution aware algorithms of the lighting network and not so much a general drawback of the distribution framework. However, even with the very limited success, some speed-up without any change to the application logic is possible.

Apart from that, one has also to take into account, that while a traditional system performs quite well in this case in terms of execution speed, it is severely limited by

wallclock seconds for	**distributed System**	**traditional VISION**
Session Setup	79.14	-
Parsing Scene	10.43	-
Lighting Setup	4.58	-
Renderer Setup	0.28	-
Init Operators	72.46	1,185.77
Render Frame	1,498.39	6,988.45
Total	**1,665.28**	**8,174.22**
	20 %	100 %



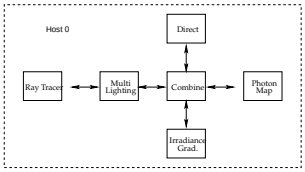


Table 8.3: On the left, the lighting network is distributed among 3 hosts. On the right, the same computations are done with the traditional single-threaded system.

the host's memory resources. Especially the PhotonMap LightOp needs to store many photons that have been shot into the scene when working with large scene descriptions. The distributed PhotonMap LightOps in this example have the memory of three hosts to their disposition. Furthermore, the initial shooting of particles is done in parallel, reducing the operator initialization time needed to at least one seventh (there are 7 processors on the three hosts), which is of great value when simulating high quality caustics.

8.2 Parallel rendering

Within the *gridlib* framework, abstraction and encapsulation is performed on several levels. Along with ensuring reentrant implementations, it is the foundation parallelization and distribution is built on. The *gridlib* uses the coarse grained message passing strategy MPI for transparent parallelization and distribution. Although the *gridlib* supports Linux on the PC, IRIX on SGI Onyx and Origin machines, we concentrate in this section on the Hitachi SR8000-F1 supercomputer architecture [KON].

Figure 8.4 shows the general data flow within the integrated simulation and visualization environment. A designated process requests the grid from the I/O subsystem and distributes it equally in terms of elements to all participating processes. Then, a refined partition is computed by all processes in parallel. Geometry or custom weighting of the elements can be taken into account (vertices $\mapsto$ grid nodes). Also, the *gridlib* can create the dual grid and compute the partition graph on it (vertices $\mapsto$ cell centroids). After performing the simulation, the partition is reused by the rendering subsystem to draw only the assigned part into a private rendering context. Note that this also distributes the memory requirement. Upon completion, all partial images are combined into the final framebuffer by a synchronized method.

Because the sizes of practically relevant datasets for CFD simulation easily exceed local memory resources, a dataset must be distributed among processing entities (PEs).

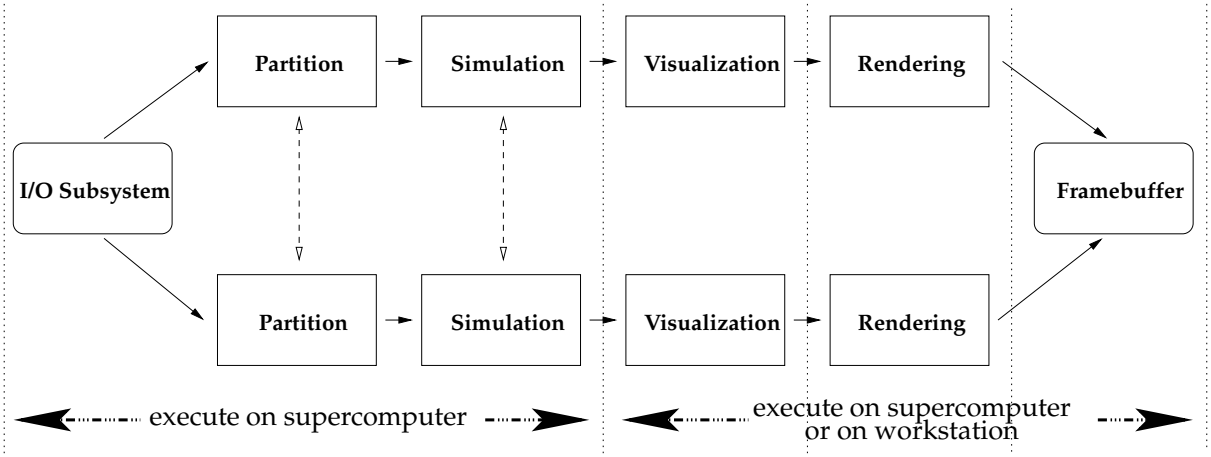


Figure 8.4: Overview of data flow for parallel rendering.

The interconnect between the PEs has much less bandwidth than local memory and is therefore a bottleneck. Consequently this communication cost dominates the total data exchange time. The *gridlib* framework processes the grid geometrically and topologically in order to build an adjacency graph in parallel. From this graph, the parallel METIS [KSK97] library computes an optimal partition which is evaluated by the *gridlib* for moving the geometry accordingly.

For running CFD solvers, the partitioning criterion clearly has to minimize the number of partition boundary elements in order to minimize the communication cost. The rendering subsystem then uses the existing partition to run the visualization and rasterization in parallel. This does not provide an equal number of triangles to render for each PE, because it depends on the visualization method and parameterization. However, transferring geometry between the PEs is too expensive both in terms of time and memory. Therefore, the visualization and rendering is performed on the same partition as the simulation to avoid delays.

8.2.1 Rasterizer performance

The following results have been obtained on a Hitachi SR8000-F1 supercomputer using a subset of 8 nodes of the machine. Each node is equipped with 9 processors with one being reserved for system activity and has 8 GBytes of local memory. The distributed resources used therefore are 64 processors with 64 GBytes of memory on 8 nodes. Although the processors of one node are sharing the memory, the coarse distribution concept of the *gridlib* performs a one-to-one mapping of processors to processes. This has the advantage of being free to place a process on any processor in any node to ensure data locality and coarse load balancing.

The rendering subsystem implementation has been evaluated for the worst case of rendering all the faces of all elements of the simulation grid. This incurs considerable

overdraw. Many visualization algorithms however output unrelated triangles, which is simulated by this approach. The triangles can be rendered immediately or can be stored in a 2D mesh. Note that a typical visualization as in figure B.2 normally has several orders of magnitude less triangles to draw than there are grid elements. Because the triangles produced by the visualization are directly drawn from the simulation grid, the renderer needs no additional memory apart from the framebuffer and some state information. All timings have been measured in wall-clock time.

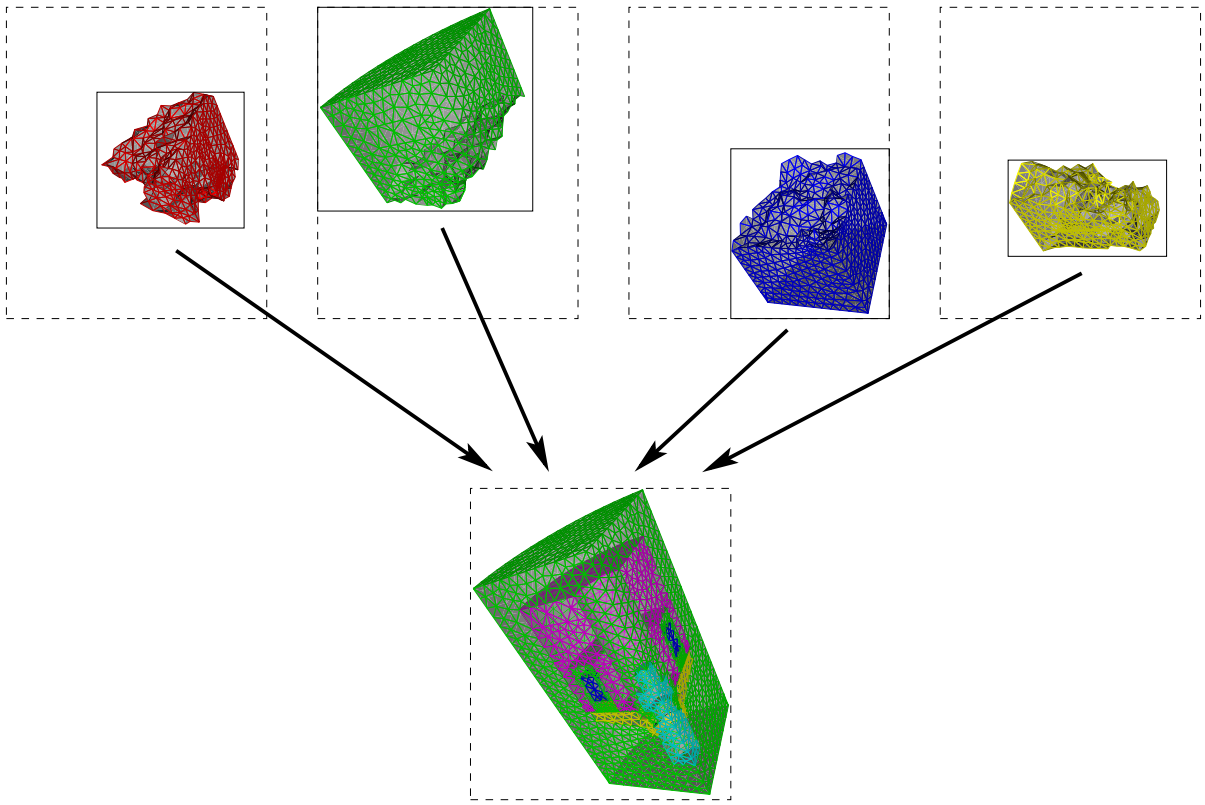


Figure 8.5: The software rasterizer within the *gridlib* rendering subsystem implements a distributed framebuffer.

Each renderer has a private rendering context. In the case of parallel rendering, the framebuffers therefore must be combined in a synchronized manner to get the final image. As shown in figure 8.5, each renderer determines the screen-space bounding box of the drawn image. When done with the visualization task, the framebuffer content and the Z-buffer of this area are transmitted to a master process along with the absolute screen-space coordinates of the bounding box. The transmission is done via blocking MPI calls, so implicit synchronization is guaranteed. The master process then combines the received partial framebuffer with its own framebuffer by performing a per-pixel overlay according to the transmitted Z-values.

Figure 8.6 demonstrates the scalability of the rendering subsystem. Almost linear

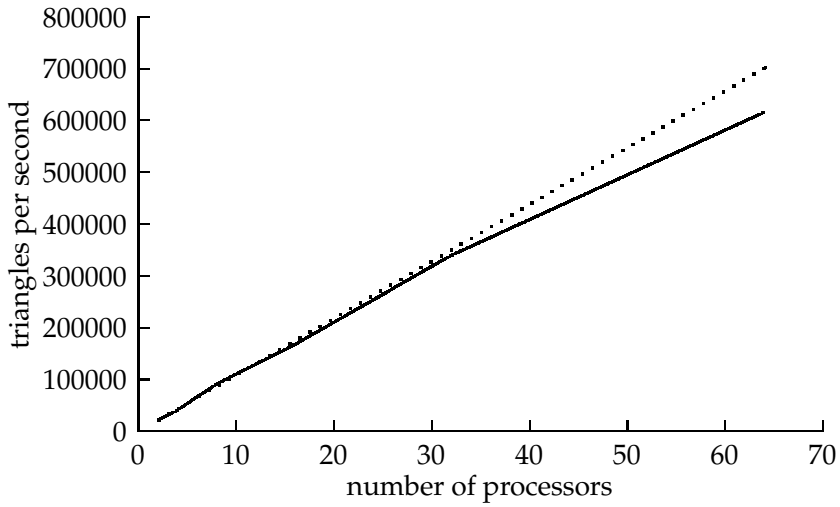
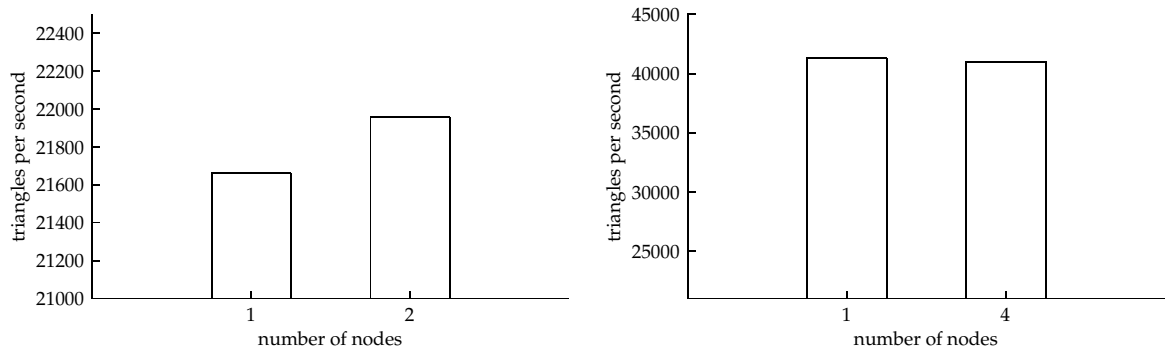


Figure 8.6: Scaling of the parallel rendering subsystem in terms of number of processors used.

speedup is obtained, although the communication volume in the Z-buffer merging stage is growing with the number of processors. The amount of pixels to transport is minimized by the method mentioned above.



(a) N = 2 (b) N = 4

Figure 8.7: Comparison of experiments of assigning N renderers to 1 node (left columns) or 1 renderer to N nodes (right columns).

The synchronization of the buffer merging stage is ensured by using blocking MPI calls. Although this seems to cause unnecessary delays, it is not a problem in practice, because the overall rendering time in most cases is typically in the range of one second, which is negligible compared to total execution times of the inner simulation-visualization loop of several minutes up to hours. Dynamic redistribution of the triangles to rasterize would incur communication overhead that slows down the whole

process considerably.

One can therefore easily afford to render the visualization on nearly every intermediate time-step while the simulation is running. The images created can be streamed back to the user interface and help to judge the simulation process to detect divergence, bad boundary setup or bad initial conditions in which case the simulation can be aborted to save CPU time.

The efficiency of this approach is shown in experiment (a) of figure 8.7, where two renderer instances running on the same node are compared to two instances running on two separate nodes. The same was done in experiment (b) using four instances on the same node and one instance on four nodes respectively. The performance differences are within the normal measurement jitter using wall-clock time. The figures clearly show that there is no time penalty for distributing the rasterizers to different nodes. In other words, the overhead of the Z-buffer merging stage is not apparent.

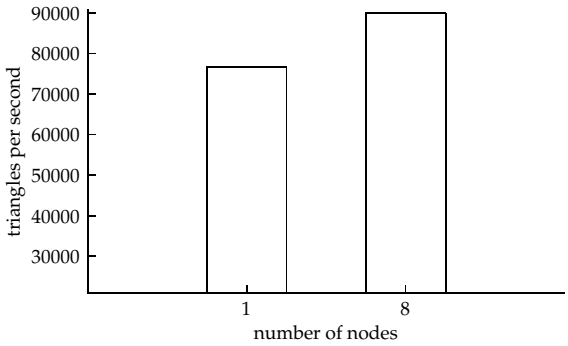


Figure 8.8: Saturation of system communication channels.

When comparing the performance of the rendering subsystem by running 8 rasterizers concurrently, a slight performance loss can be observed in the case of running all rasterizers on a single node (figure 8.8 left column). This is due to suboptimal MPI support for intra-node operations on the Hitachi. In contrast, when distributing one rasterizer to each one of the 8 nodes, no performance penalty is observed (figure 8.8 right column). Normally, one would expect that inter-node operations are more expensive than intra-node operations. Therefore, additional profiling has been performed on an SGI Onyx machine (shared memory architecture), which does not exhibit this behavior and thus clearly identifies the shared memory communication to be the bottleneck of the Hitachi supercomputer. This also justifies the approach not to compute a separate partition of the grid for rendering. The communication time incurred by it easily exceeds the total time spent for rendering. As the coarse distribution strategy was to employ MPI, the bottleneck is easily circumvented by placing only one rasterizer on each node. The rasterizers performance is sufficient for response times of one second and no intra-node communication is induced that may slow down the simulation code.

8.2.2 Optimization

As mentioned earlier, for the performance tests above, the worst case of rendering all faces of all elements individually has been presumed. This can be compared to OpenGL rendering triangle by triangle in immediate mode. Clearly, a lot of optimizations are possible if the renderer is allowed to work on a larger 2D triangle mesh, which can be

compared to rendering OpenGL triangle-strips in retained mode. This comes at the expense of additional memory requirement for storing the 2D mesh and some acceleration structure. The *gridlib* rendering subsystem features several standard optimizations for 2D triangle meshes. When using them, the software rasterizer has a peak performance of up to 91,000 smooth-shaded triangles per second on a single CPU.

8.3 Ray tracing in hardware

As, at the time of writing of this thesis, there is currently no existing implementation of a ray tracer using FPGA technology, some related work should be reviewed first. It can be divided into two main categories: Using a lot of hardware to make the ray tracer interactive or using hardware to satisfy special requirements of the rendering algorithm.

One of the first papers about real-time ray tracing is from Muuss et al.. They implemented an image-space parallel ray tracer on a 96 processor SGI PowerChallenge [ML95]. A similar approach [PMS+99] is the real-time ray tracer of Parker et al.. Recent research by Slusallek et al. has shown the importance of exploiting coherence inherent in the ray tracing approach [WSBW01]. Their implementation uses 7 dual-processor PCs. They validated the theoretical statement, that ray tracing is faster than triangle rasterization for large numbers of triangles [WSB01]. All approaches share the property of being pure software implementations that have been tuned to platform specific hardware capabilities. They have a large number of general-purpose processors and (nearly) unlimited memory resources at their disposition. The obtained frame-rates are closely related to the time required for preprocessing.

G page 145

For the special case of volume rendering by ray casting, Pfister et al. presented in 1999 the first single-chip real-time rendering engine for consumer PCs, the *VolumePro* board [PHK+99]. It has been preceded by similar university research projects [SB94, Kni96]. The rendering performance is possible because of the embarrassingly parallel nature of the shear-warp algorithm for orthogonal projection that is used. Recently, Pfister et al. have proposed an extension of the VolumePro approach, the *RAYA* architecture [PK01]. It extends the voxel-based ray casting to a full ray tracing solution for both voxel and geometric primitives and may provide programmable shading. Up to now (October 2002), there is no working sample available.

In contrast to the highly specialized VolumePro board, Mai et al. have proposed the multi-purpose massively parallel and hierarchical *Smart Memories* architecture [MPJ+00]. It features general-purpose RISC processors with local RAM and high-bandwidth interconnects tiled onto a single chip. Although this approach is not explicitly targeted for ray tracing, the huge numeric power and the parallelism available make this concept an ideal candidate for image-space partitioned coherent ray tracing.

Other related work comes from Advanced Rendering Technologies. The commercially available *RenderDrive* [ART99] uses several full custom design chips to build a hardware unit that serves as computation device for offline, non-interactive, high-

quality rendering applications. The device is not targeting interactive frame rates, but concentrates on fully programmable, *RenderMan* compliant shading. It uses an array of custom designed ASICs with RISC core and 32 floating-point units each, which makes the system quite expensive.

All of the above approaches are based on RISC processors or try to alleviate numerical demanding computations (VolumePro). The biggest drawback however is their dedication to a specific task. Changing the algorithm to obtain customized results can completely invalidate their run-time behavior. The RenderDrive and the Smart Memories architecture seem to offer the biggest free reserves and flexibility. Their enormous numerical and memory resources suggest brute-force approaches.

Styles et al. in contrast have shown, that current generation FPGA chips (→ section 3.3) offer the possibility to develop customized graphics applications that use a set of simple commands to execute custom graphics functions on the FPGA [SL00]. They implemented a triangle rasterization system that delivers frame-rates equivalent to software rasterization. Current generation graphics cards however outperform the FPGA by far. We therefore propose to use the FPGA as a graphic co-processor for non-rasterization tasks, like ray tracing.

This approach seems to be the most promising, and Purcell et al. have tried to solve the problem using consumer graphics cards [PBMH02]. They did not manage to get the system running on current hardware, as some vital instructions are missing. The results they obtain from a simulator however show clearly that a gradual convergence between ray tracing and the feed-forward hardware rasterization pipeline is possible.

8.3.1 System Overview

The system presented in this section is targeted to provide a complete autonomous ray tracer on a single chip. The host application is able to obtain a rendered image by just providing camera settings and the scene description and issuing a start command. Upon completion, the image can be read back from the FPGA or it can be directly transmitted to the standard framebuffer for display, overlay over an image or for textured surfaces created by the (triangle rasterization) graphics card.

In order to be as reusable as possible, the ray tracer is responsible for the scene handling. The host application should not be burdened with preprocessing. The drawback of this approach is, that the scene must fit in the local memory of the FPGA. This also means that the ray tracer is not optimized for interactive frame rates. It must allow rather general object descriptions as our goal is to demonstrate the general utility of the FPGA technology for rendering high-quality images.

Hardware Layout

The ray tracer consists of four major building blocks as shown in figure 8.9:

- The **ray generator** reads the camera setup and screen resolution from the local

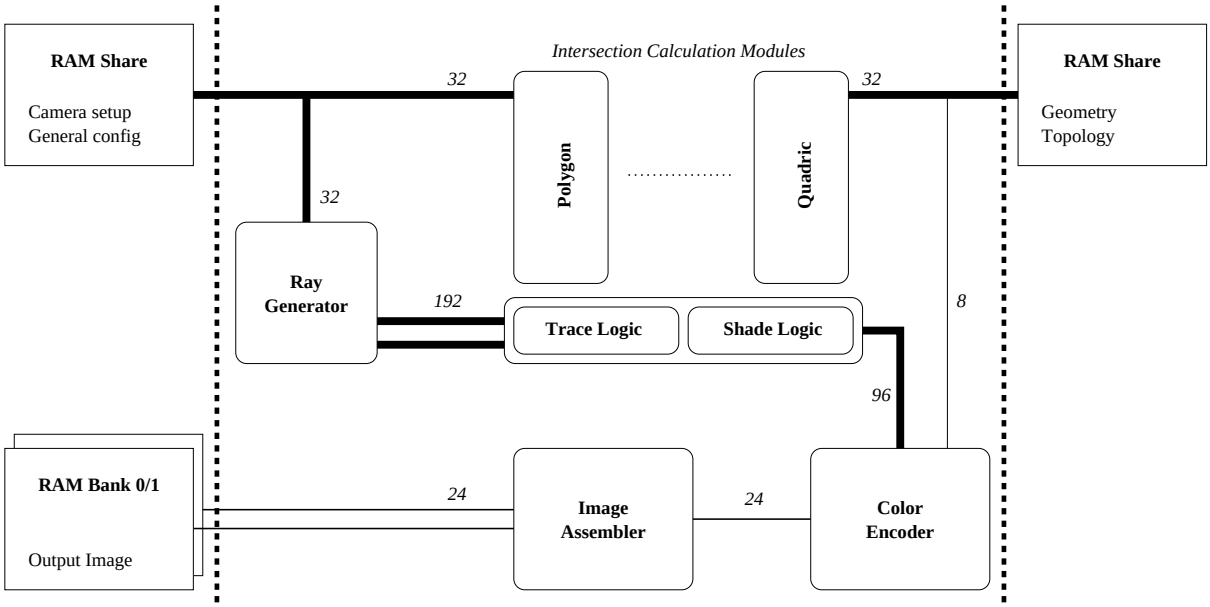


Figure 8.9: The overall chip layout of the FPGA ray tracer. The annotated numbers display the Bit-width of the busses.

RAM. It generates all primary rays for one frame and sends them down the rendering pipeline channel. The generator also applies some normalization transformations and consistency checks to the camera setup. Upon completion of the frame, the generator immediately loops and reads the camera setup for the next frame.

- The **tracing and shading logic** is the core part of the ray tracer. It consists of a general skeleton algorithm that represents the program flow common to all ray tracers. It has associated modules that perform the specific tasks of intersection calculation and shading for some specific object or surface type.

- The **color encoder** reads one pixel at a time in the internal pixel format and converts it to some common representation. We currently support RGBα and YUV.

- The **image assembler** places the data obtained from its input in the appropriate local framebuffer location on the local RAM. Because transferring a completed frame to the host or to the graphics card also takes a fair amount of time and requires to be able to lock the transfer buffer, the image assembler uses double-buffering to ensure good throughput: Upon completion of one frame, the image assembler releases the buffer and signals the event to the host. Now the buffer can be transferred while the second one is assembled.

The subsystems are coupled by synchronous peer-to-peer busses (see below). Each subsystem operates independently and loops indefinitely. The synchronization with the

host application is done indirectly by the image assembler: It will block until the host clears the buffer-ready signal of the buffer in question. This in turn will halt the preceding subsystems output until the pipeline is completely filled up.

The ray generator, the color encoder and the image assembler subsystem are straight forward implementations that process one request at a time. The tracing and shading logic implements a skeleton algorithm for intersection testing and shading. It uses small modules that actually perform the intersection calculation for a specific object type. As the processing of one depth-step of a ray involves several intersection calls for the primary and secondary rays, the tracing and shading logic takes advantage of running several modules in parallel. Note that this means true parallelism in terms of replicated hardware. In the following pseudocode for the tracing and shading logic, parts marked by `par {...}` are executed as parallel threads. The statement following these blocks is executed not until all threads have finished. Parts marked `seq {...}` will execute sequentially.

Note that the proposed architecture has no scene traversal unit. In order to make use of advanced acceleration structures, such a subsystem is necessary. This currently prohibits the use of the FPGA ray tracer for rendering large scenes and explains the slow rendering times for moderately complex scenes in section 8.3.3. Advanced scene traversal has not been implemented because of the limited chip surface.

Internal Communication

The major subsystem blocks of the ray tracer communicate over several synchronous busses, called *channels*. In figure 8.9, the channels are drawn as thick lines. The attached numbers show the width of the bus in Bits. A channel supports bi-directional communication.

The channels serve as a means of decoupling the subsystems. Each part can run independently. Because all subsystems share the same clock signal, the overall performance of the ray tracer is only limited by the slowest subsystem, which is the one that needs the largest number of clock cycles N to complete one request. Because of the pipeline structure of the whole system, the ray tracer can produce one pixel every N clock ticks after an initial delay of $4 * N$ clock cycles to fill the pipeline. Note that the clock rate of the chip therefore does not match the output frequency of the processing channel. The main layout therefore is not a pipeline, strictly speaking (→ section 2.2).

As accessing local memory on the PCI card can be done only once per clock cycle, multi-ported on-chip RAM registers are used to prefetch data or to store intermediate results. The RAM also can be accessed only once per clock cycle, but multiple RAM registers can be accessed in parallel. The local memory on the PCI card in contrast provides only four banks that can be used in parallel. The algorithms have been examined carefully to understand at which instant certain data is accessed and prefetching from the local RAM registers has been implemented, so a complete data structure can be read in a single clock cycle by the algorithm when it is needed. The prefetch has been determined

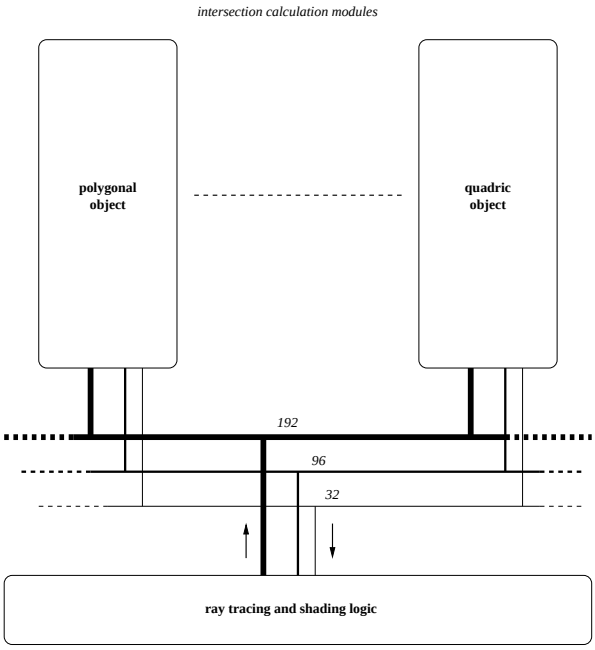


Figure 8.10: Calculation of intersection point and surface normal is modular. The annotated numbers display the Bit-width of the busses.

experimentally and is written explicitly directly in the algorithm. Simple data structures like vertex positions are stored by using a different memory bank for each dimension and can therefore be read in a single clock cycle at any time.

As the tracing and shading logic implements only the basic skeleton algorithms, it provides uni-directional busses for communication with modules that provide intersection and surface normal calculation. Figure 8.10 shows the busses with their Bit-widths attached. For each primitive supported, there is a special module that can provide the requested information. Note that the modules are independent of each other and therefore run as stand-alone engines. The pseudocode 8.3 connects a module with the tracing and shading logic.

External Communication

The ray tracer can be controlled by the host application through a small set of commands. A library that implements these commands has been written. It manages the data transfer to the memory on the PCI card. The API concept is built after the OpenGL state machine: The camera setup of the ray tracer is done similar to camera setup in OpenGL and defining the scene to be rendered is similar to defining an OpenGL display list. There are state variables for modifying certain properties of the ray tracer, especially for starting and stopping rendering. The library translates all configuration settings and uploads the data to the appropriate RAM locations on the PCI card.

The API also features commands for defining where the output of the ray tracer should go. Normally, the library stores the rendered image internally. It can be queried by the host application at any time. Alternatively, the library can set up a DMA transfer directly from the FPGAs PCI card to the graphics card. The transfer is executed automatically as soon as the FPGA has completed one image by using the PCI bus mastering capability of the card. This concept makes the FPGA ray tracer a valid graphics co-processor that effectively takes work-load off the main CPU.

8.3.2 Implementation

The ray tracer has been developed using Celoxicas Handel-C [Cel01] hardware programming language. It is a high-level language with a strong resemblance to C.

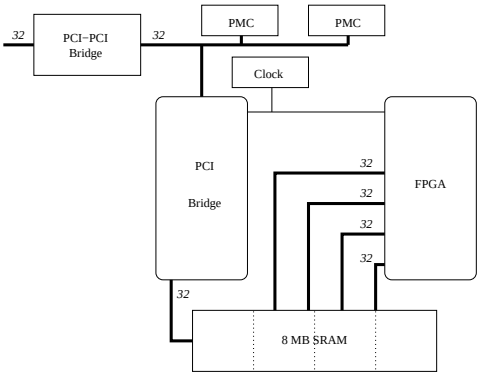


Figure 8.11: The FPGA is mounted on a PCI card with I/O Controller and 8 MByte local memory.

The program is translated into an intermediate format which can be compiled to obtain code that runs within a simulator, or it can be converted into a netlist in EDIF format. The FPGA manufacturer Xilinx [Xil01b] provides place and route tools to create the final bitmap image from the netlist. The bitmap is then uploaded to the FPGA from the host application to actually program the chip. This process does not involve creating VHDL. Although some performance loss may be associated with this approach, it has a much steeper learning curve for computer graphics people.

The Handel-C language supports both replication and parallelization of code for running several instances of one thread in a way that is relatively easy to control and synchronize. The internal communication channels of the ray tracer make heavy use of these concepts.

As shown in figure 8.11, the FPGA is mounted onto a PCI card that is equipped with 8 MByte of SRAM. Both the host and the FPGA can access the RAM in 32 Bit word-size address mode. The integrated PCI bridge provides functionality for locking the RAM and can act as bus-master on the host PCI bus. The RAM supports transfer rates up to 25 MHz. It is organized into four banks, so the theoretical maximum transfer volume is $4 * 25\text{MHz} * 32\text{Bit} \approx 400\frac{MByte}{sec}$. While this is valid for the FPGA, the host communication must pass through the PCI bus, which has a theoretical peak performance of $\approx 133\frac{MByte}{sec}$. The communication volume with the host application therefore needs to be kept at a minimum. Currently, dynamic loading of scene parts from the host is not supported. The scene to be rendered has to fit into the PCI card RAM.

G
page 145

Numerical Issues

In order to make programming and reuse of existing algorithms easy, the ray tracer accepts input values in the standard IEEE 754 single-precision floating point format. Furthermore it uses floating point calculus for most of its internal algorithms. Although this sounds very attractive, there are several problems associated with it.

Floating point calculations cannot be executed directly, as the FPGA has no dedicated floating point units. The basic arithmetical operations of signed addition and subtraction, multiplication, division, square root, float to integer and integer to float conversion have to be encoded explicitly. Although this is true for pure integer arithmetic too, encoding the algorithm of the operation can be done using considerably less CLBs of the FPGA compared to their floating point counterparts. This results in smaller circuit delays for the operation in question, which in turn defines the limiting frequency at which the circuit can operate correctly. One standard approach to obtain higher operating frequencies is to break down the operation into simpler steps and pipeline them. This however in turn will increase the number of CLBs that are necessary. The basic arithmetic floating point routines therefore use a large portion of the chip surface, because in order to execute higher-level numerical routines in parallel, there must be several instances of each of them. The implementation of the FPGA ray tracer has 6 multiply, 2 divide and 12 add engines that are used by all parts of the pipeline in a coordinated manner.

8.3.3 Results

The system is based on a consumer 1200 MHz Athlon PC with *n*VIDIA GeForce 3 Ti200 graphics card running a Linux 2.2 kernel. We use a Xilinx Virtex 2000E FPGA chip [Xil01a] that supports operating frequencies up to 100 MHz. It is mounted on a PCI card as described in the previous section. Uploading a FPGA programming image and personalization of the chip takes about 150 ms.

For the external communication, bus master DMA transfers are used. On the test system, transfer rates of $49\frac{MByte}{sec}$ for reading from the PCI card and $38\frac{MByte}{sec}$ for writing to it have been measured. Direct transfers to the framebuffer of the main graphics card provide $50\frac{MByte}{sec}$ of bandwidth. This is enough for displaying 1024×768 pixels at 16 frames per second in RGBα-mode. Thus, the transfer of the rendered images does not present a bottleneck at current.

Figure 8.12 shows the rendering of a quadric test object. The image is 200×200 pixel and displays at ≈ 0.7 frames per second. It employs two directional light sources, classic Phong shading, recursion depth 1 and no transparency.

The ray tracer supports polygonal objects with up to 140 faces and 3 or 4 vertices per face. The face polygons may be concave. As the scene has to fit into the RAM, the scene size is limited to 1024 polygonal objects. For demonstrating the rendering of analytical surfaces, the ray tracer also supports quadrics of order 2. A quadric is defined by $Ax^2 + Bxy + Cxz + Dx + Ey^2 + Fyz + Gy + Hz^2 + Jz + K = 0$ and can therefore be

stored within 10 floats. This allows more than 52000 quadrics to be stored in the RAM. Because of the general nature of the ray tracer implementation using floating-point arithmetic and the demand for high numerical performance when rendering quadrics, we use 94% of the FPGA resources ("slices"). The missing scene traversal unit leads to slow rendering times even for moderately complex scenes. The teapot scene (128 polygons) in figure 8.13 takes $\approx$ 2 minutes as does the 5 quadric scene because the quadric intersection module has a much longer latency because of the numerics involved.

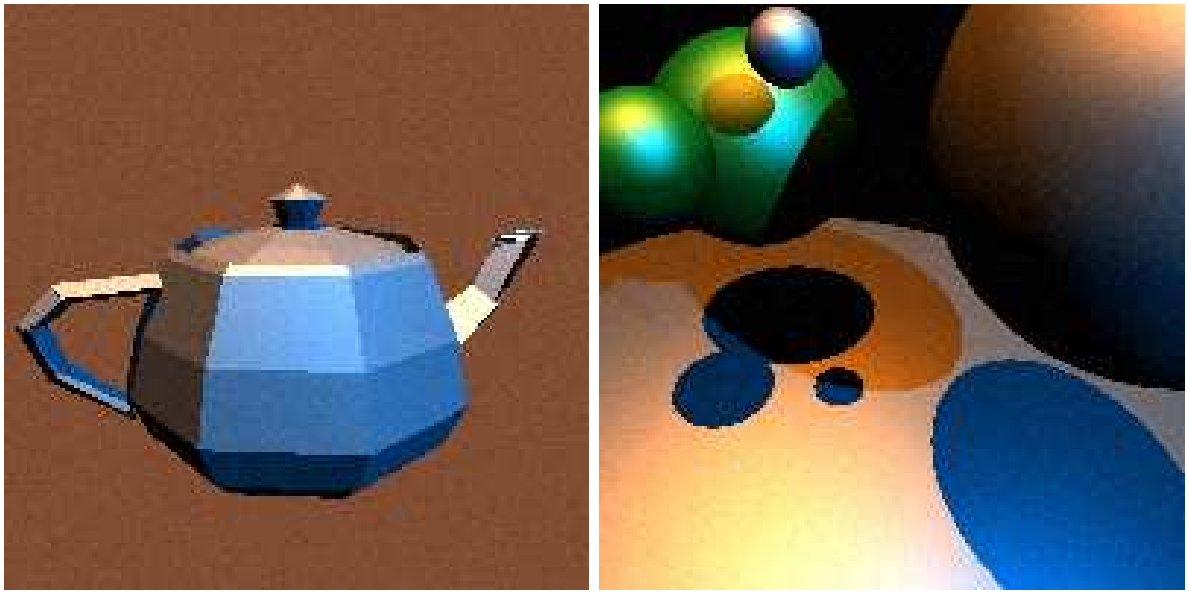


Figure 8.13: A polygonal teapot and a quadrics scene rendered with the FPGA ray tracer.

Color plate B.4(a) shows the routed layout of the ray tracer with selected connections of the ray generator subsystem in red color. The main connections and the intersection module channels of the tracing and shading logic are marked in green. Note how the modular concept also helps the place and route tools to optimize path lengths and localization of resources. On the right hand side, figure B.4(b) shows a visualization of the degree of local connectedness of the CLBs of the FPGA. Again, the two subsystems are clearly distinguishable. The color encoder and the image assembler subsystem are straight forward implementations that have not enough complexity to show up.

Because of the highly complex floating-point arithmetic, our implementation has a rather deep nesting of circuits. The maximum depth is 61 logic levels and requires a chip clock delay of $\approx$ 140 ns. This limits the maximum operating frequency to 7 MHz. The set of floating point macros provided by Celoxica is suboptimal and turned out to be the limiting resource of the ray tracer. They compute the floating-point operation within one clock cycle at the expense of very deep nesting. However, for production code, there exist other commercial floating point cores [NL01] with very high efficiency both in terms of function performance and space requirements. They are implemented as pipelines with shallow logic and offer operation frequencies of up to 165 MHz.

The ray tracer concept presented in this section is rather general. It therefore can serve as a basis for several rendering tasks. Much higher framerates are possible by restricting the object descriptions to simpler primitives like spheres and triangle lists.

The FPGA layout is kept intentionally as modular as possible to enable processing in a pipelined fashion, easy maintenance and using it for educational purposes. As each module obtains its input from channels and sends it output to other channels, it is easy to write a small test framework for it for debugging purposes. Using the FPGA simulator provided by the Celoxica IDE, the channels can be attached to files on the harddisk which makes examination of the results with third-party tools easy. The module concept offers a further privilege: Many FPGA circuits support partial configuration and read-back. One can take advantage of that for exchanging the intersection and surface normal calculation modules at runtime without destroying all the state information in other modules that would occur when performing a chip personalization, i.e. a complete reconfiguration by uploading a whole FPGA netlist image.

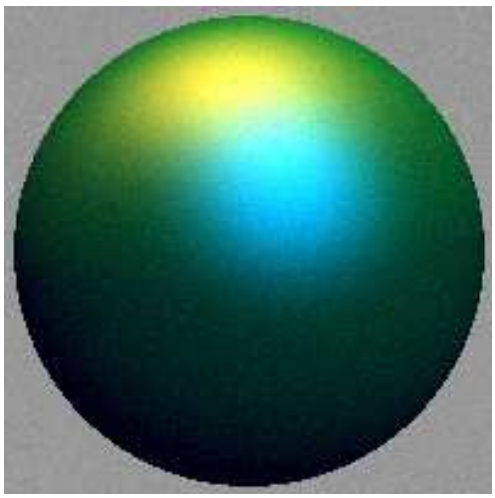


Figure 8.12: The quadric test scene (→ color plate B.3(a)).

In color plate B.3(b), a sample application of the ray tracer for generating a high-quality mirror image has been implemented. The ray traced image is put into a texture which is applied to a quadrilateral within the test scene that is rendered using OpenInventor.

```
T_and_S {
 while (true) {
  par {
   seq {
    sRay = reflect(last_ray,last_t,last_n);
    for (allObjects) {
     num = intrCh.select(moduleList[pRay.primType]);
     par(num) {
      intrCh.send(sRay);
      s = intrCh.param();
     }
     (s > 0) ? color=shade(last_ray,last_t,last_n);
    }
    outputCh.send(color);
   }
   seq {
    pRay = inputCh.waitfordata();
    for (allObjects) {
     num = intrCh.select(moduleList[pRay.primType]);
     par(num) {
      intrCh.send(pRay);
      t = intrCh.param();
      n = intrCh.normal();
     }
     (t > 0 && t < min_t) ? min_t=t; min_n=n;
    }
    par {
     last_ray = pRay;
     last_t = min_t;
     last_n = min_n;
    }
   }
  }
 }
}
```

Listing 8.2: Pseudocode for the tracing and shading logic of the FPGA ray tracer.

```
intersectObj {
  while (true) {
    resetLocalState();
    ray = intrCh.waitfordata();
    ... algorithm goes here ...
    intrCh.setParam(t);
    intrCh.setNormal(n);
    intrCh.release();
  }
}
```

Listing 8.3: Pseudocode for an intersection module of the FPGA ray tracer.

Part IV

Conclusion

9

SUMMARY

Mit dem Wissen wächst der Zweifel.

— Johann Wolfgang von Goethe

Numerical simulation and scientific visualization and rendering are complementary tools for modern fluid dynamics research. The size of the result data and the comparatively limited external bandwidth of current high performance supercomputers strongly suggests integrated processing. Application developers are therefore faced with many requirements the program has to meet to deliver performance that can be rated as adequate compared to the theoretical capabilities of current hardware. This thesis contributes a classification of distribution and parallelization strategies for scientific high-performance computing and demonstrates their usage with several implementations.

The first part of this thesis presented a categorization of several complementary strategies for distribution and parallelization that attack the problem on different levels. The strategies have been examined and evaluated within several implementations. For building a supporting framework for distribution and parallelization for application developers, a combination of the presented strategies must be used. The coarse-level strategies lay the foundation for distributed computing. As they follow a specific distribution concept, they get deeply embedded into the application logic. The decision for using the one or the other strategy depends on latency, bandwidth and bridgeable distance the application has to cover.

For writing effective algorithms, the intermediate strategies provide a simplified view of the underlying hardware to the programmer. As these strategies are to be used in conjunction with the coarse-level strategies, they concentrate on parallelization of more or less local algorithms. To this end, they emphasize the importance of data handling and collaboration of internal parts of the application. In contrast to the coarse-level strategies, they never interfere with the application or distribution logic. They are ab-

straction tools for program development and maintenance.

The fine-level strategies deal with efficiency aspects. They heavily depend on data structures and processing operations. They are not specific to a certain application, but provide general tools for exploiting current hardware capabilities. Because of the architecture of current computer systems, their use is inevitable if progress in future processor development should be utilizable by the application or library.

The second part of this thesis presented the integrated simulation, visualization and rendering framework library *gridlib* that has been implemented for evaluating the above strategies. The *gridlib* is an outstanding contribution to supercomputing, as it delivers the enabling technology for efficient handling of grand-challenge problems. It provides an object-oriented software infrastructure for common grid-based numerical simulation problems on unstructured, hybrid grids. At the same time, it abstracts from the underlying hardware details. Its three-tier architecture provides generalized memory handling that can be customized for third-party simulation codes, object-oriented abstraction of geometry elements, edges and vertices and a mesh container that serves as central access point to a set of elements, edges and vertices that form a grid. The mesh container also explicitly supports algorithmic abstraction and computation services. For the application developer, the *gridlib* offers high-level client subsystems for common tasks like I/O, visualization and graphical display. The functionality of the subsystems can be used to implement new high-level object-oriented simulation codes. In section 5, an overview of applications using the *gridlib* was given. Using the services and grid management functionality, their development time was very short. Thanks to the features of the *gridlib*, all applications are portable without special precautions of the programmer.

The third part of the thesis presented several new contributions to distributed and parallel computing. Most of the applications that have been implemented for evaluating the distribution and parallelization strategies are built on the *gridlib* framework because of its integration functionality on supercomputers and desktop systems alike.

The distributed lighting networks system (→ section 8.1) examined the possibility of applying the coarse distribution strategy of CORBA to build a distributed framework for an existing rendering system without interfering with the application logic. The only requirement was that the existing subsystem decomposition must allow separation of address spaces. Using multiple layers of wrapping objects allowed to distribute objects and parallelize computation through subsystem replication transparently to the developer of the actual computation classes. This sets the distributed lighting networks system apart from other parallel ray tracing applications that embed the distribution logic deeply into the core classes, which is therefore highly visible to the programmer. In section 8.2, the opposite approach was taken for building a high performance parallel rendering subsystem. The whole design has been dedicated to the coarse distribution strategy MPI for obtaining maximum data transport performance. It is available as a high-level client subsystem within the *gridlib* framework. It allows the generation of images of intermediate solutions of a running simulation at virtually no cost. Both im-

plementations demonstrate the flexibility of the coarse strategies and emphasize at the same time the programming effort necessary for employing them efficiently.

Sections 7.1 and 7.2 examined intermediate distribution and parallelization strategies to verify the promised advantages. Like it has been pointed out, the strategies do not interfere with the application logic but offer better flexibility and parallel operation for the programmer. They are valuable tools to enhance the efficiency of the application. The interactive 3D stream player presented in section 7.1 is the first working tool for interactive rendering of time-depended scalar volumes for an unlimited number of time steps. It benefits from the abstraction introduced by the pipeline strategy that allows to exchange a module implementation if hardware resources can be exploited. The local exact particle tracing method presented in section 7.2 leverages previous theoretical work to a useful visualization tool thanks to sophisticated parallel pre-processing using the OpenMP strategy.

The fine-level strategies have been studied in section 6.1 and 8.3. Both applications are examples to show the general potential of the strategies. While SIMD processing is already available on most PC architectures, the FPGA hardware approach needs additional support from hardware manufacturers to become common practice in the future. Both examples show the importance of considering fine-level strategies when designing an application that should be able to use nowadays available hardware efficiently.

Pure high level language program optimization fails to exploit current hardware capabilities because of weak compiler optimization engines and assembler backends that lack the extended processor instruction sets. The examined Lattice Boltzmann flow solver application demonstrates how SIMD instructions can be utilized without deteriorating the programming patterns of the high level language. The achieved results present a speedup factor of three without changes to the hardware or the high level program.

The presented autonomous FPGA ray tracer is the first complete and working single-chip solution on low-cost hardware without expensive additional floating point units. Recent developments with PC graphics cards also point in this direction and make the developed FPGA ray tracer a valuable first proof of concept for future applications of programmable graphics hardware.

10

FUTURE CHALLENGES

Why waste time learning
when ignorance is instantaneous ?

— *Hobbes*

10.1 Integration of functionality

Like it has been pointed out in section 4.1, nowadays problem sizes in technical simulation demand huge computing facilities. The strategies that have been presented in this thesis show the possibilities of integrating simulation, visualization and rendering, which is essential to bridge the gap between local memory resources and bandwidth of external communication channels.

The crucial point that must be considered for future research therefore is to examine the compatibility of data structures for simulation and visualization, their adaptability and performance for a given hardware environment. Shareable distribution and parallelization models must be investigated and implemented as library functionality that is transparent to the application developer. For achieving good levels of abstraction within such large frameworks and applications, compilers for object-oriented programming languages must be implemented on supercomputer machines, which is not the usual case nowadays. Successful integration of simulation, visualization and rendering will become one of the key points on such architectures, because none of these disciplines will be able to deliver adequate processing times when implemented and optimized on its own.

10.2 Flexible SIMD processing

The encouraging results for SIMD processing for Lattice Boltzmann methods, as presented in section 6.1, should be exploited for other technical simulation codes. Future

research therefore should investigate the SIMD parallelization potential of the code in question. Especially codes operating on structured grids are good candidates for this strategy.

Simulation codes for unstructured grids offer conceptual much less opportunities for SIMD processing because of the varying neighborhood information that has to be computed explicitly. However, the major part of the SIMD performance comes from the ability of the processor to access the fast caches efficiently. As enlargement of the caches is too expensive and the main memory access speed will stay limited for current architectures, Intel has proposed to leverage the processor performance by *hyper-threading* [Int02]. This can be understood as SPMD processing on a single CPU, because the processor is not limited to perform exactly the same instruction at the same time as with the SIMD concept, but executes the code independently like multiple CPUs with the advantage of a shared cache. This technology can help tremendously with performing simulations on unstructured (potentially hybrid) grids, as multiple neighboring cells can be processed in parallel while efficiently using the cache. The functionality of hyper-threading can be made available transparently to the programmer, by integrating it into an OpenMP implementation.

10.3 Integrated FPGA technology

Because the ray tracing concept has been studied for quite a while now, there is a huge amount of different algorithmic approaches and optimizations for each of the subtasks. However all of them have been evaluated as software implementations. In order to categorize and analyze the performance potential, their applicability to hardware implementation must be thoroughly investigated. The FPGA-based ray tracer presented in section 8.3 does not implement any acceleration technique other than trivial bounding box calculations. This is directly related to investigations in integer-based algorithms and precision considerations, as this will free chip resources occupied by the complex floating point macros. The liberated space can then be used for an advanced scene traversal unit to get decent rendering speed for moderately complex scenes.

Further acceleration of the ray tracer through image-based approximation techniques must be investigated. The *holodeck* [Lar98] technique for example produces a new image by manipulating the previous image and complementing it with newly traced rays in critical regions. However this may give noticeable artifacts and it will demand additional FPGA surface and RAM for the image-based part. Also asynchronous communication of subsystems must be investigated to account for the different algorithmic complexities, as modern FPGA chips support more than one clock domain.

The main future challenge of course is an integration of programmable FPGA chips on consumer graphics cards. They would complement the high speed rasterization operations with custom programmable data processing. The recently introduced *n*VIDIA GeForce 3 chip set and its successor GeForce 4 already have some basic programmable

data manipulation functionality for the autonomous keyframe interpolation feature of the *vertex shader* [NVc] concept. The FPGA ray tracer then would have a direct connection to the framebuffer RAM, which presents the ideal opportunity to use it as a high-quality texture generator. The game industry would also profit from it as it would allow a much more flexible creation of 2D effects than current concepts like the *pixel shader*. Furthermore, the FPGA will profit from the better performance of the AGP bus host connection. Hybrid rendering techniques like proposed recently by Stamminger et al. within their *corrective textures* [SHSS00] walk-through system will benefit from this.

Part V

Appendix

A

GLOSSARY

ACE The adaptive communication environment library, → section 1.3.1.

barrier synchronization A synchronization point is inserted into the program. For the barrier, all processes of a group reaching this point are halted there. When the last process of the group reaches the point, all processes resume. For a mutual exclusive (mutex) synchronization, only one process is allowed to enter the synchronization point at a time.

binding of implementations If a function is called as a subroutine, only the parameters, the return value and the starting address of the subroutine need to be known to the caller. This can be exploited for delaying the actual implementation of the subroutine. If the starting address and the parameter information is reserved for the subroutine, the code for its implementation can be loaded and bound to the starting address on-demand.

broadcasting These are collective communication operations (→ section 1.2.2) that occur frequently in parallel and distributed programs. Consequently, communication libraries like → *MPI* offer optimized system calls for them. Broadcasting copies one variable to all participating processes. Gathering collects the variable's values from all processes, computes a result from all received values and stores it with the initiating process. The reduction operation that is performed can be configured. The scattering pattern distributes the components of a local vector to all processes such that each process gets some configurable part of the vector.

cache A memory cache is a small piece of high speed memory that is often clocked at the same frequency than the processor I/O channels. This makes it extremely expensive, but allows the processor to quickly load its registers. When accessing some data in main memory, a whole cache line is loaded into the cache, because the probability that the processor will access adjacent memory locations is good. Writing cache-aware algorithms therefore needs to account for data locality, so the majority of the operations will not request additional cache lines, which would force the cache to write and read from main memory permanently, destroying the speed advantage.

ccNUMA cache coherent NUMA → *NORMA*

compiler pragmas Nearly every high-level compiler consists of several processing steps. The first step is a pre-processor that is responsible for collecting all source code necessary for the current translation unit. The second step is the code translator, that builds an abstract syntax tree which is encoded into executable binary format by the third step. Pragmas are compiler control instructions in the source code, that are not removed by the pre-processor and therefore can control the actual code translator engine.

CORBA Common Object Request Broker Architecture → *OMA*

endianess Because of internal design of the processor circuits, the binary representation of integer values can be different. The endianess describes the position of the most significant Byte (MSB) within multi-Byte integer variables. Big-endian machines store the MSB first, little-endian machines store it last.

event de-multiplexing Events arriving on a single channel are identified and separately forwarded to the appropriate recipient.

gathering → *broadcasting*

IDL Interface Definition Language → *ORB*

interoperability The ability of implementations of a communication library from different manufacturers to work together.

IOR Interoperable Object References → *ORB*

lightweight classes In object-oriented languages, a class encapsulates both data and functions into a unique object. Inheritance plays an important role for declaring class relations. However, everything that is added to a class declaration complicates the storage layout of the implementation. A lightweight class has only data members and no inheritance relation. It may have some simple (inline) access methods. Its memory layout therefore is exactly the concatenation of its data members.

mesh subdivision Each element of a mesh can be refined into multiple smaller elements of the same type or of other types. This subdivision process can be performed on all elements (uniform) or only on some elements (adaptive). In the case of adaptive refinement, hanging nodes need to be corrected to get a new correctly closed mesh.

middleware Library that builds an abstraction of some underlying service, but is not intended to provide end-user functionality, → *CORBA* section 1.3.2.

MIMD multiple instruction, multiple data → *SISD*

MISD multiple instruction, single data → *SISD*

MPI The Message Passing Interface library is the most prominent representative for distributed supercomputer applications, → section 1.2.2.

mutex synchronization → *barrier synchronization*

NORMA no remote memory access; Variations of the → *MIMD* architecture class describing supercomputer hardware available today (→ introduction to part one).

NUMA non uniform memory access → *NORMA*

object retention In a distributed environment, it may be advantageous not to destroy remote objects immediately after the client has dropped the reference on the object. The → *CORBA* standard allows to keep the server object and reuse it for further invocations. This behavior is controlled and configured by a system service.

OMA Object Management Architecture; Definition and implementation of an object-oriented → *middleware* communication library, → section 1.3.2.

OMG Object Management Group → *OMA*

ORB The Object Request Broker. Part of the → *CORBA* communication library.

parallel file systems In order to amend the disk I/O throughput, the data is written to multiple harddisks simultaneously by the file system. Each disk holds a fraction of the files ("file-striped") or a fraction of the data of each file ("block-striped").

portal culling When rendering a partial view of a scene, some objects may be totally outside the viewing frustum. They are removed ("culled") from further processing. The same can be done with parts of the scene that are visible only through a small window in some geometry. This portal obscures all geometry that lies behind its limiting walls. Therefore, portal culling can considerably reduce the number of objects to render.

processor instruction In order to build a generally useful programmable active device, a processor is structured into several parts (→ *von Neumann architecture*). The processor instructions therefore contain besides arithmetic operations also control commands for data flow, signaling for subsequent parts and error conditions and special internal operations like register-based SIMD operations. As the processor has to orchestrate the collaboration of all these parts, many clock cycles are needed that cannot be used for the actual numerical operations.

processor instruction pipeline As the processor repeatedly fetches the next instruction from memory and executes it, this process is implemented as a (at least) three stage pipeline. The three parts run independently and therefore allow the processor to execute one complete operation within one clock cycle on average. The pipeline is split in instruction fetching, instruction decoding and execution. Most often, an additional branch prediction unit allows to keep the pipeline filled even if there are conditional jumps in the code.

programming design pattern When working with large object-oriented systems, many programming patterns are recurring regularly. The standard approach to solve such problems is to employ design patterns that are classified according to scope and purpose [GHJV95]. Using such a pattern is highly recommended, as they are proven concepts, offer well known semantic and pragmatic aspects, complexity and limitations.

PVM The Parallel Virtual Machine communication library offers the master–slave collaboration model, → section 1.2.1.

reentrant implementation The implementation can be executed by several processes simultaneously without side effects.

scattering → *broadcasting*

shear-warp For rendering 3D voxel data using 2D texture slices, the rays casted at the volume pass it in some specific angle respective to the front face. In order to trace straight rays, the texture slices are stretched and sheared, and the resulting image is warped to the image plane. This gives the same result as tracing the rays at specific angles through the volume, but can be implemented much faster.

SIMD single instruction, multiple data → *SISD*

SISD single instruction, single data; Classification of parallel computer architectures according to Flynn [Fly72] dependent on data streams and available → *processor instructions* (→ introduction to part one).

SPMD single program, multiple data; Variation of the → *SIMD* architecture class with the relaxation that the simultaneous code execution is performed on the program level, not on processor instruction level (→ introduction to part one and section 1.2.2).

strong typing High level programming languages assign a unique type to each data structure or object. The validity of applying an operation or accessing internal structures can only be checked, if the type information cannot be hidden or counterfeited by the programmer. Strongly typed languages like C++ enforce this automatically, while compiler for weakly typed languages leave it to the responsibility of the programmer.

TAO The ACE ORB → *OMA*

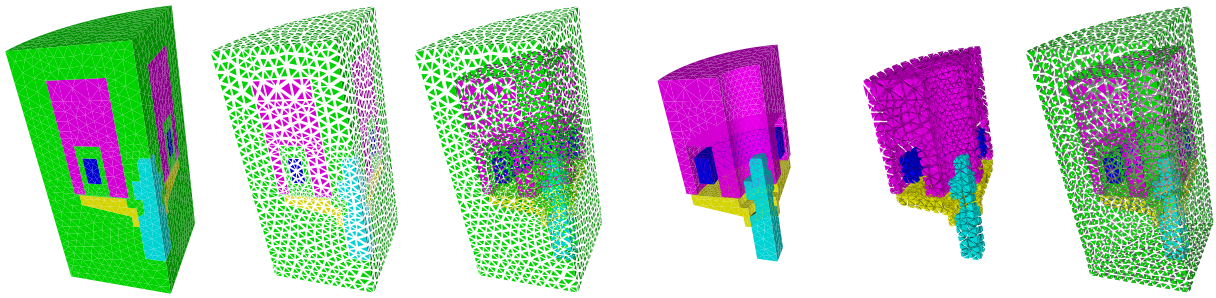
UMA uniform memory access → *NORMA*

von Neumann architecture The concept of a universal computer has been introduced by Burks, Goldstine and von Neumann in 1946 [Kla89]. The architecture of the system is structured into four functional blocks: the control, arithmetic, memory and I/O unit. In order to process an algorithm, a sequence of → *processor instructions* are decoded and forwarded to the corresponding functional block by the control unit. The program is fetched from the memory unit. Branching instructions enable to build non-linear instruction sequences. The control unit has access to internal state of the other functional blocks. Branching can therefore be conditional. In order to allow higher level processor instructions, the control unit can execute micro-programs.

Z-buffer When a triangle is rasterized into a framebuffer, the Z-values of the projected pixels are stored in the Z-buffer. Deciding whether a pixel should be drawn over an existing pixel then is simple by comparing the new Z-value to the buffer content. Because the Z-buffer algorithm is very simple, it is implemented in hardware.

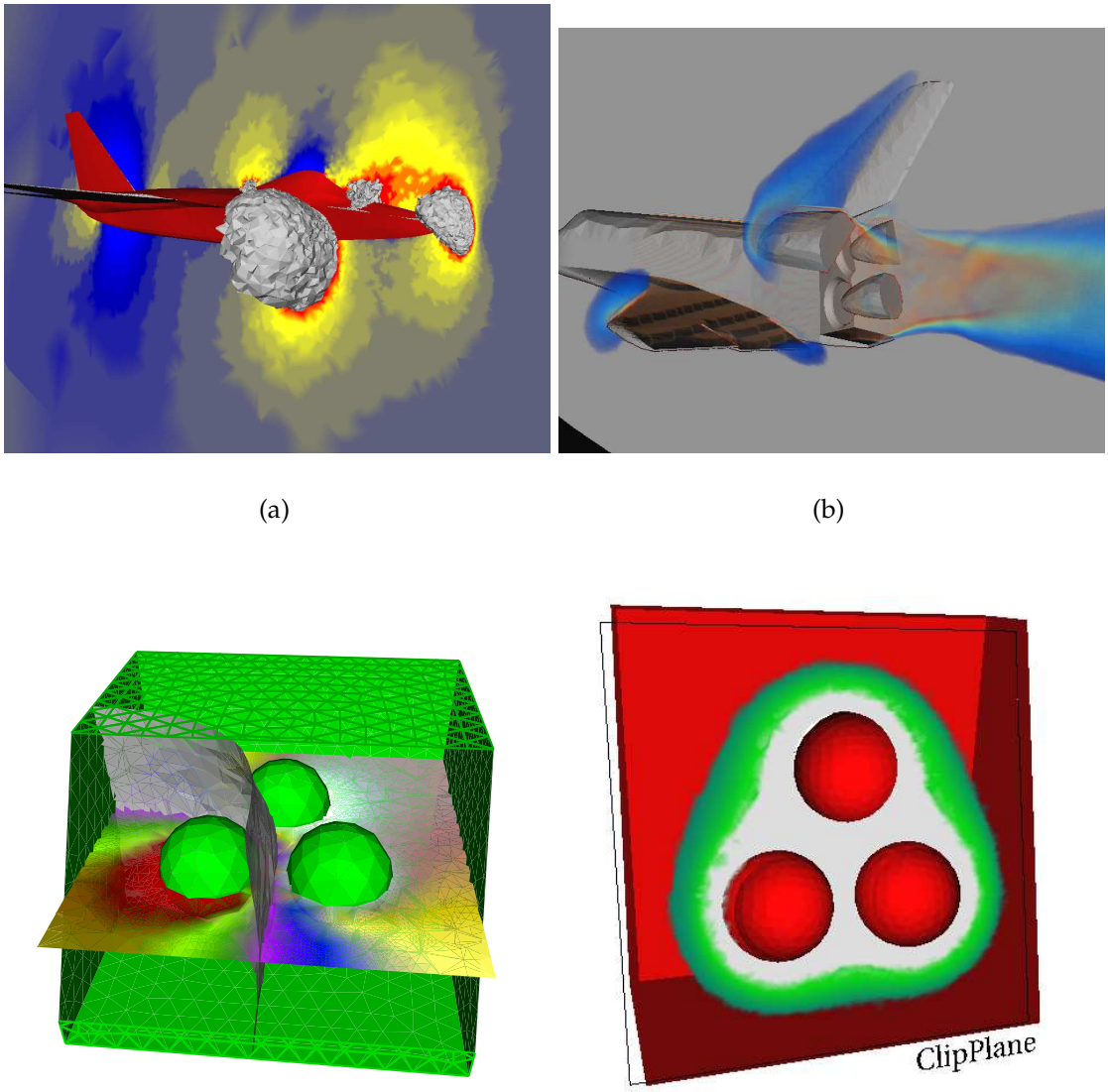
B

COLOR PLATES



(a) (b) (c) (d) (e) (f)

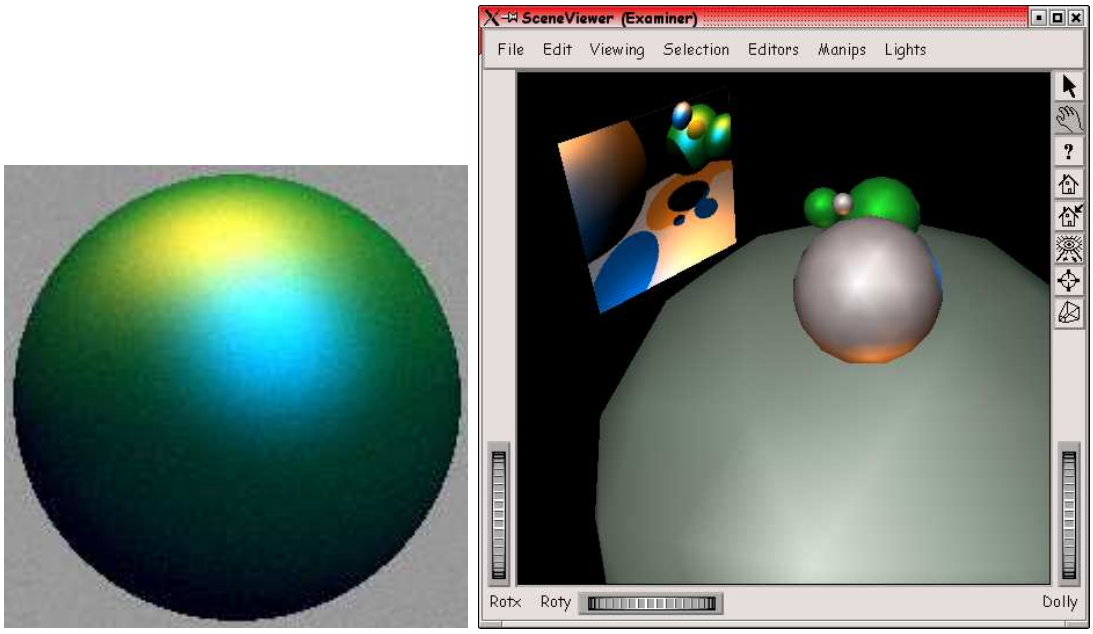
Plate B.1: The rendering subsystem has several rendering modes for close examination of detail or interactive rendering of general shape. The images above show a partitioned dataset. Each partition can be turned on/off independently. Each mode or a combination of modes can be applied to each partition separately: (a) surface (b) surface faces of adjustable size (c) faces of any boundary (d) surface of partition (e) volume elements of adjustable size (f) double-sided with greyed-out back faces



(a) (b)

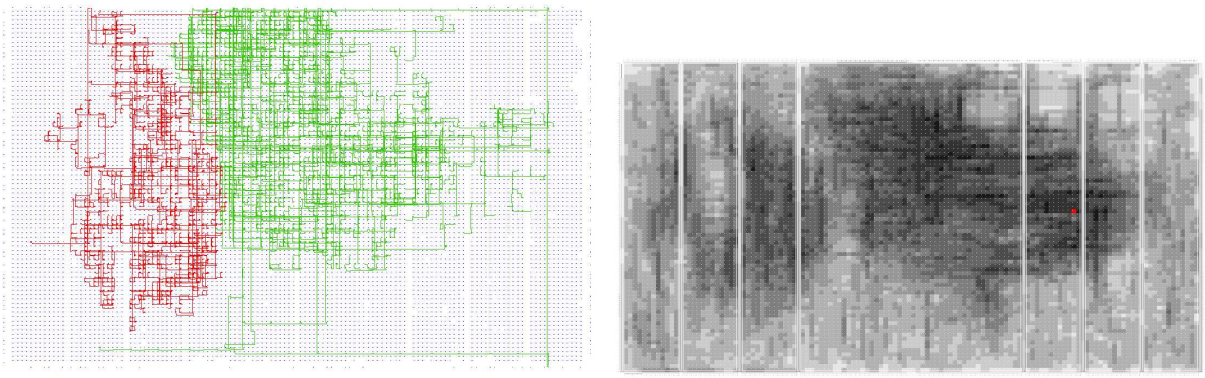
(c) (d)

Plate B.2: Examples of possible direct visualization: (a) an airplane with color-mapped density slice and pressure isosurfaces, (b) the space shuttle with air velocity mapped by direct volume rendering, (c) an electrostatic simulation showing some special scalar value as an isosurface and a color-mapped slice through the unstructured simulation grid, (d) is the same dataset now with a simulation of heat dissipation by direct volume rendering. The data set is cut open by clipping planes.



(a) (b)

Plate B.3: Figure (a) shows a single quadric which renders at 0.7 frames per second. Figure (b) is a snapshot of an application where the output of the ray tracer is put into texture memory for display as a mirror image within a rasterized scene.



(a) (b)

Plate B.4: Image (a) shows the routed layout of the ray tracer with selected connections of the ray generator subsystem in red and the main connections and the intersection module channels of the tracing and shading logic in green. Image (b) shows a visualization of the degree of local connectedness.

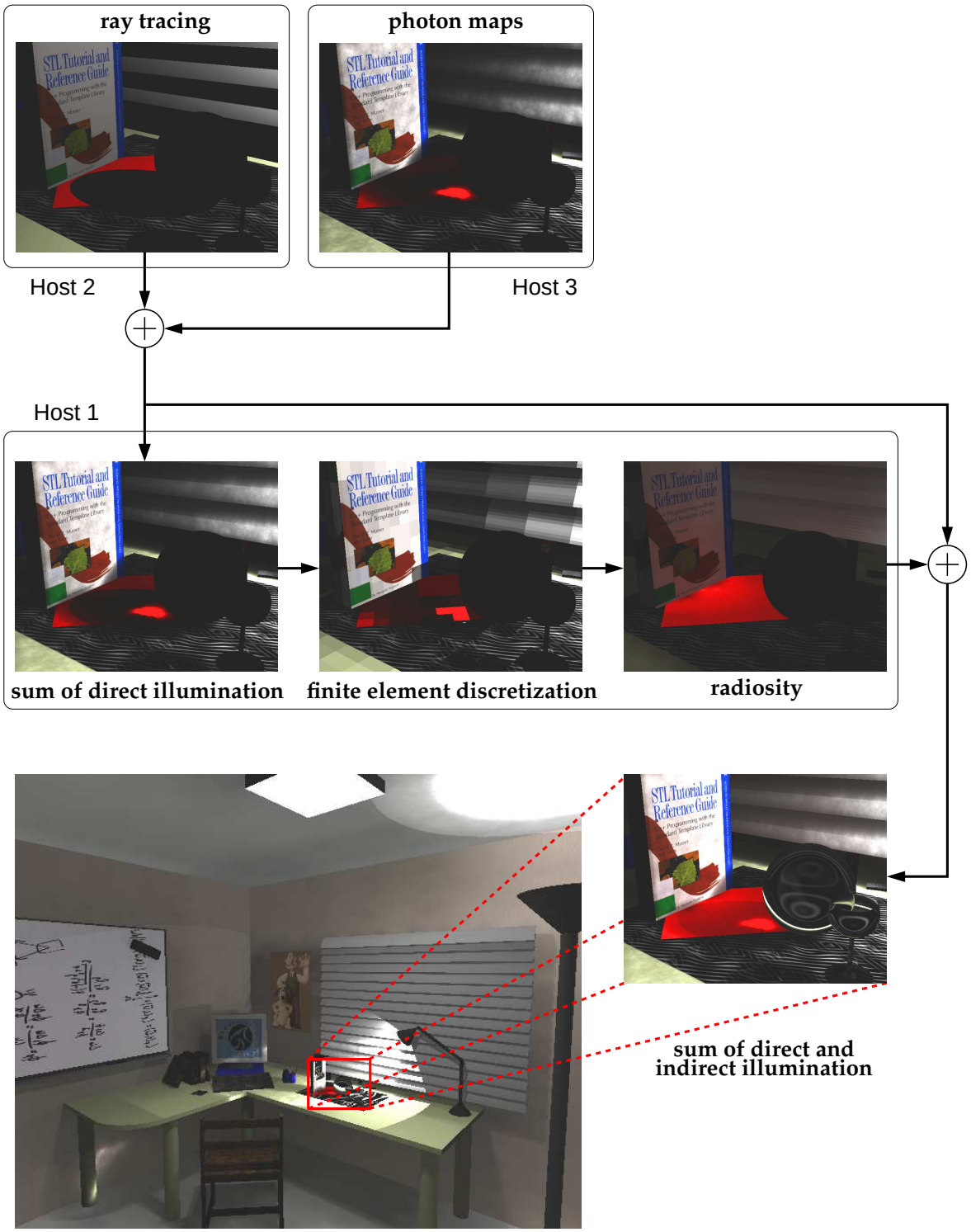
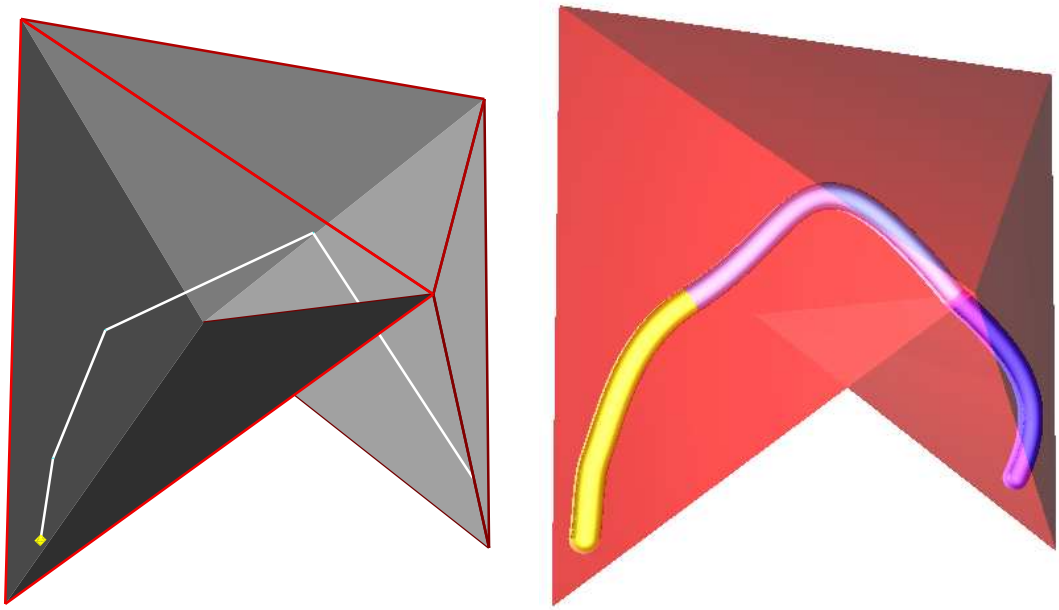
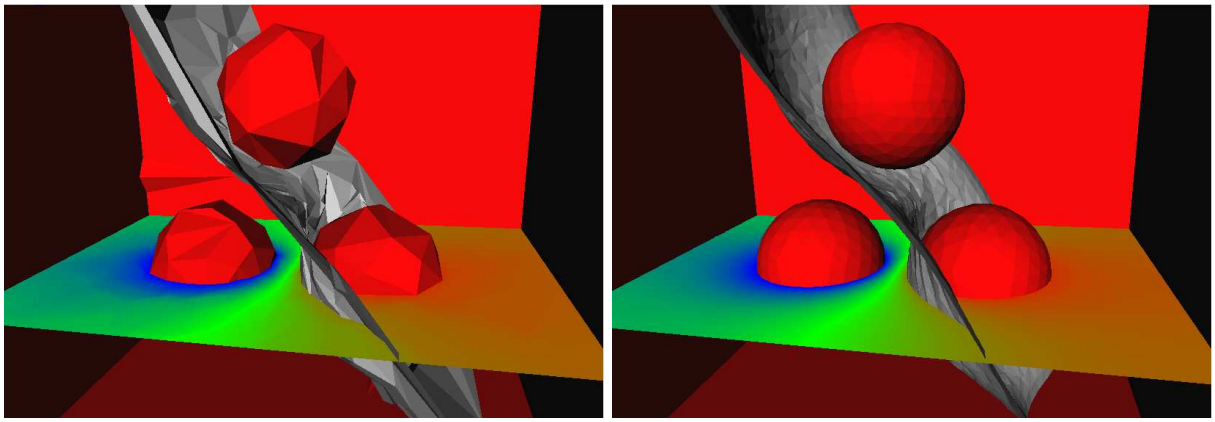


Plate B.5: Data flow and intermediate results computed by a distributed lighting network performing direct, indirect and caustic illumination through different LightOps.



(a) (b)

Plate B.6: High-quality visualization using the local exact integration method for particle tracing. In (a), the trajectory is rendered with OpenGL. The smooth ray traced curve in (b) is colored differently in every segment computed.



(a) (b)

Plate B.7: Progressive transmission of a dataset. The isosurface visualization is coupled to the progressive grid and therefore also progressive. Image (a) shows the coarsest level, image (b) the final refined result.

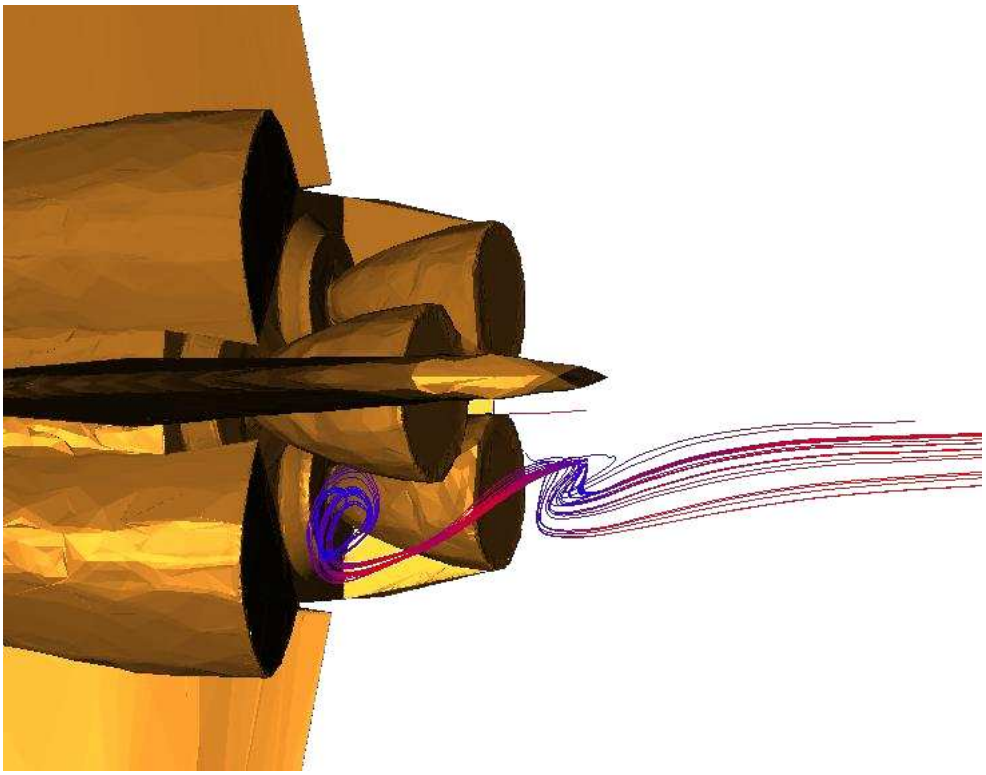


Plate B.8: Shuttle in the wind tunnel computed with the local exact method.

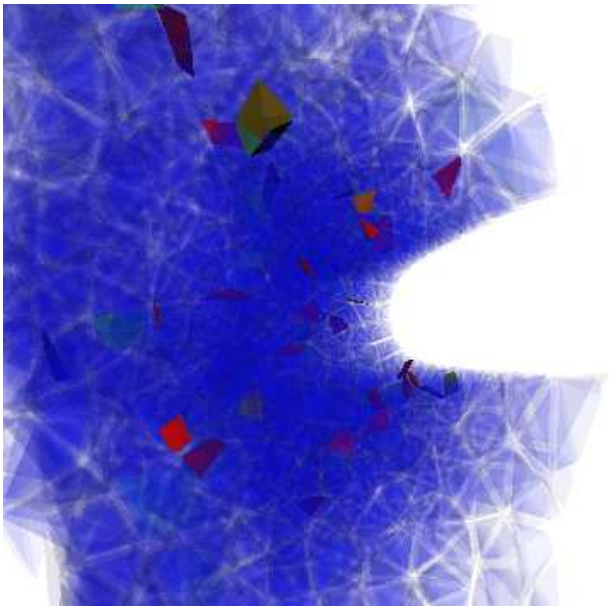
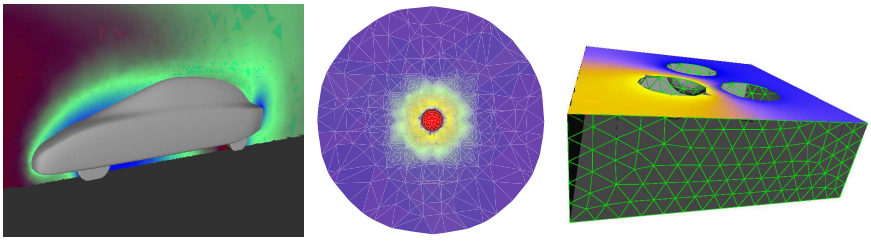
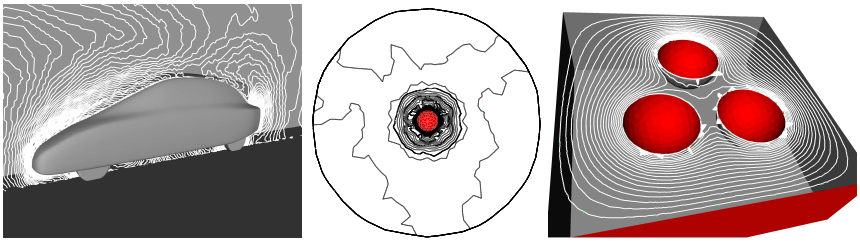


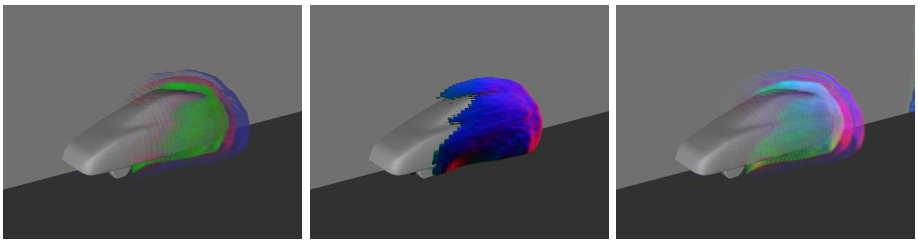
Plate B.9: Visualization of tetrahedra quality: red ones are bad.



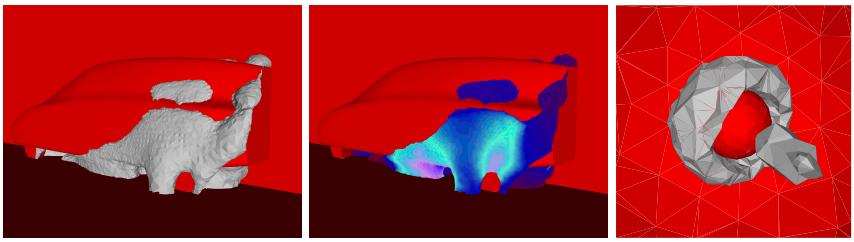
Planar slices. Scalar values can be mapped onto the slice using a color table. The table can be modified interactively.



Contour lines on a planar slice. The lines are generated as true geometry and therefore display in correct perspective. The iso-value of a line is annotated.



Direct volume rendering by regular resampling. The voxel volume can be rendered with hardware support. Interactive color table mapping and gradient shading for visualization of tiny structures and isosurfaces is available.



Isosurfaces of any scalar value. The surface can be shaded for good spacial impression or another scalar value can be mapped onto it using an interactive color table.

Table B.1: The *gridlib* offers several visualization algorithms for displaying simulation results.

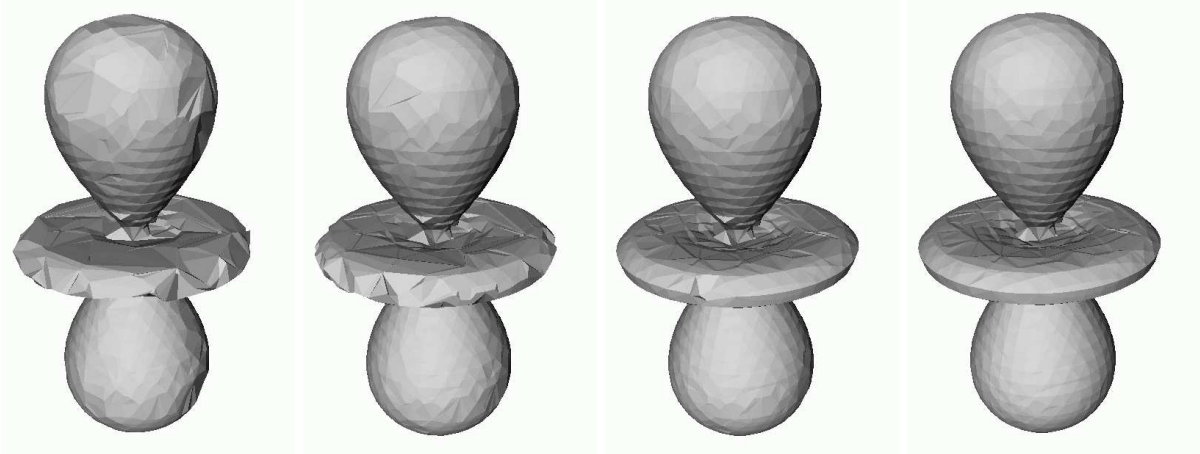


Plate B.10: Progressive isosurface extraction, from left to right using 5%, 10%, 16% and 30% of the original number of elements of the dataset.

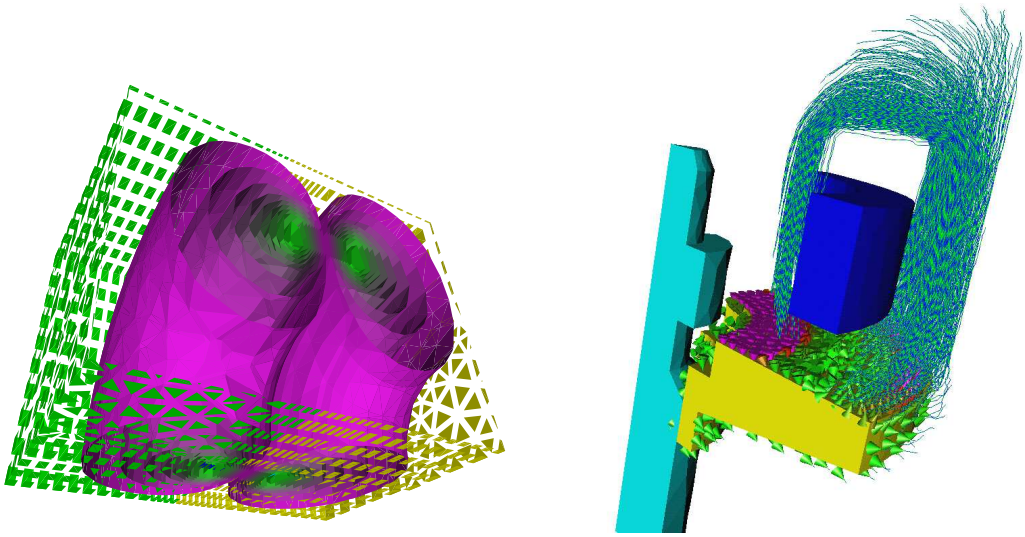


Plate B.11: Left: Fast isosurface extraction on unstructured grids. The concentric surfaces represent a specific iso value over time. Right: Visualization of a magnetic field simulation.

C

INTERFACE INHERITANCE IN *gridlib*

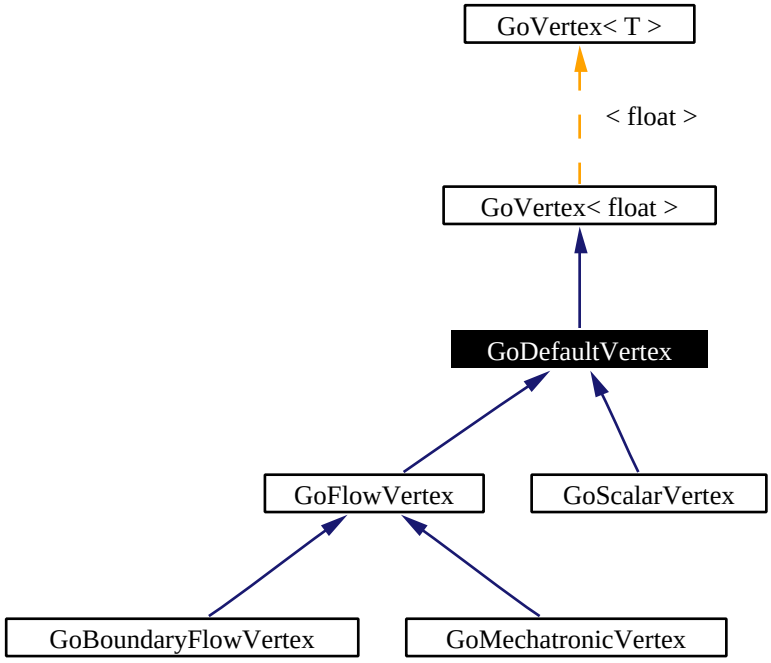


Figure C.1: Inheritance diagram for *gridlib* vertices.

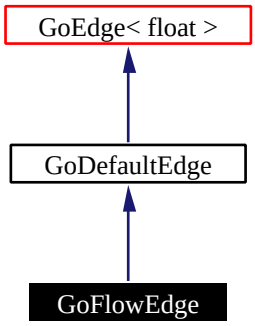


Figure C.2: Inheritance diagram for *gridlib* edges.

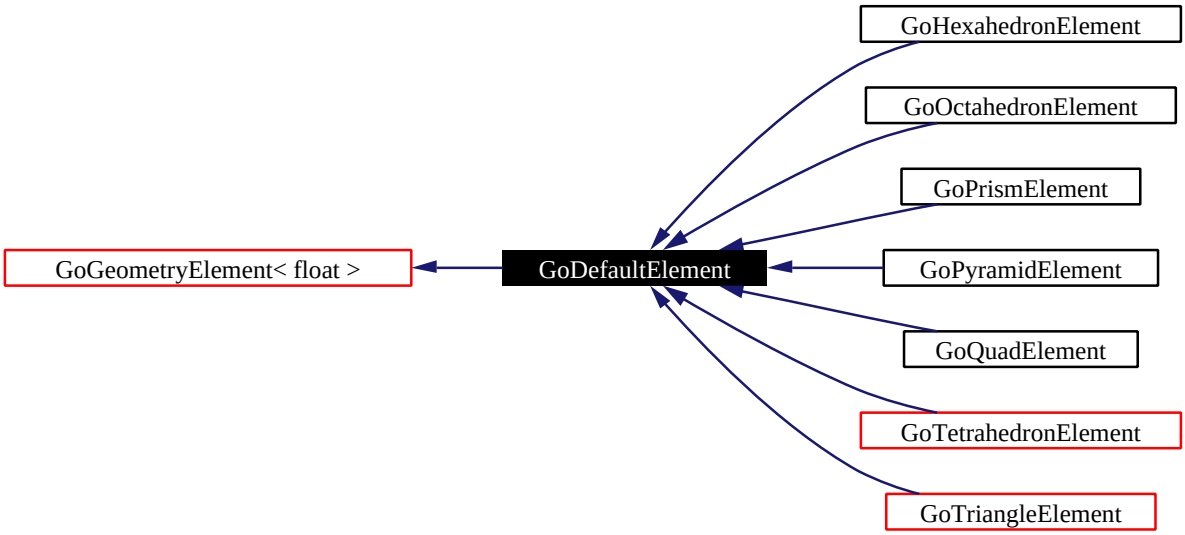


Figure C.3: Inheritance diagram for *gridlib* geometry elements.

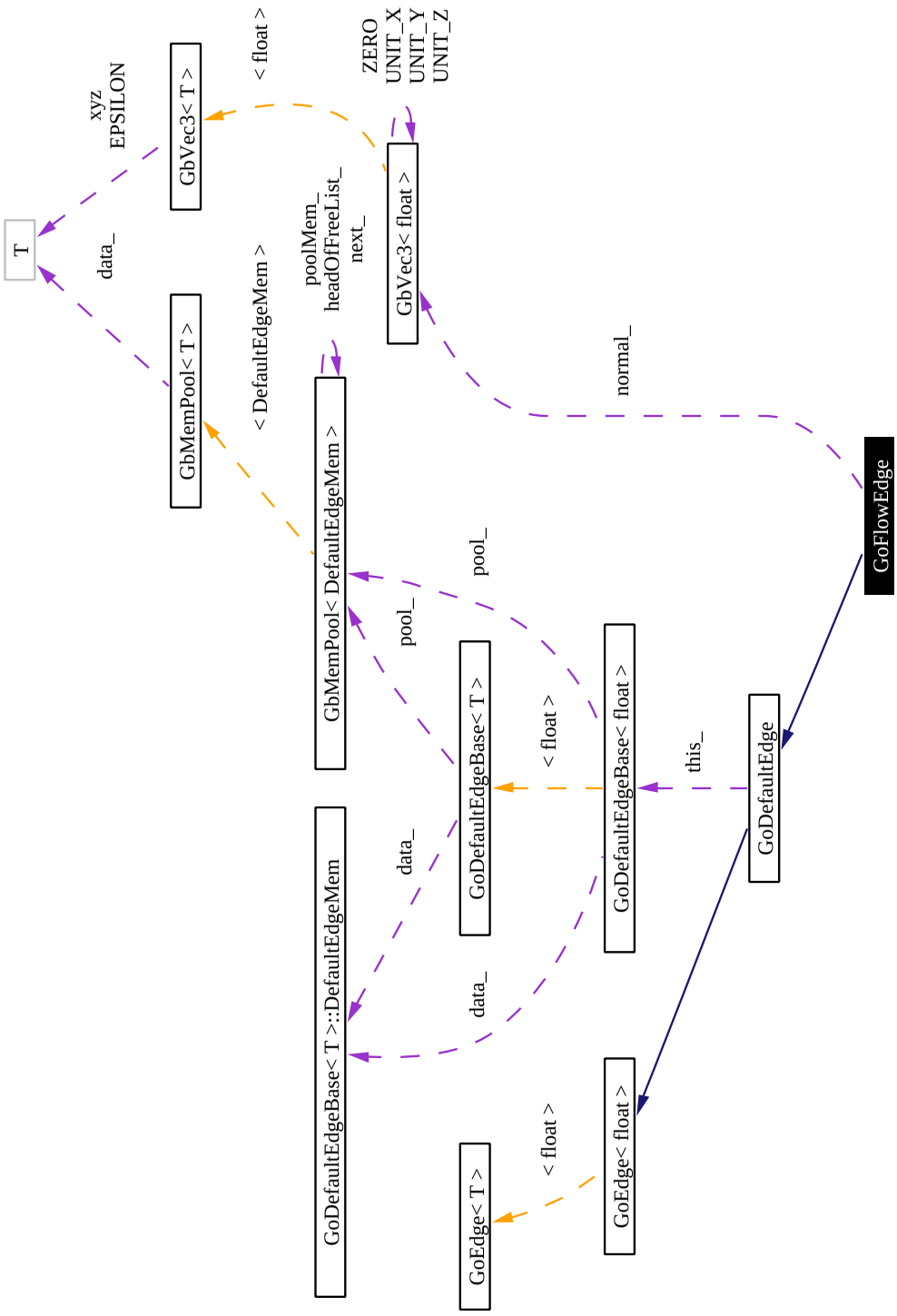


Figure C.4: Collaboration graph for *gridlib* edges.

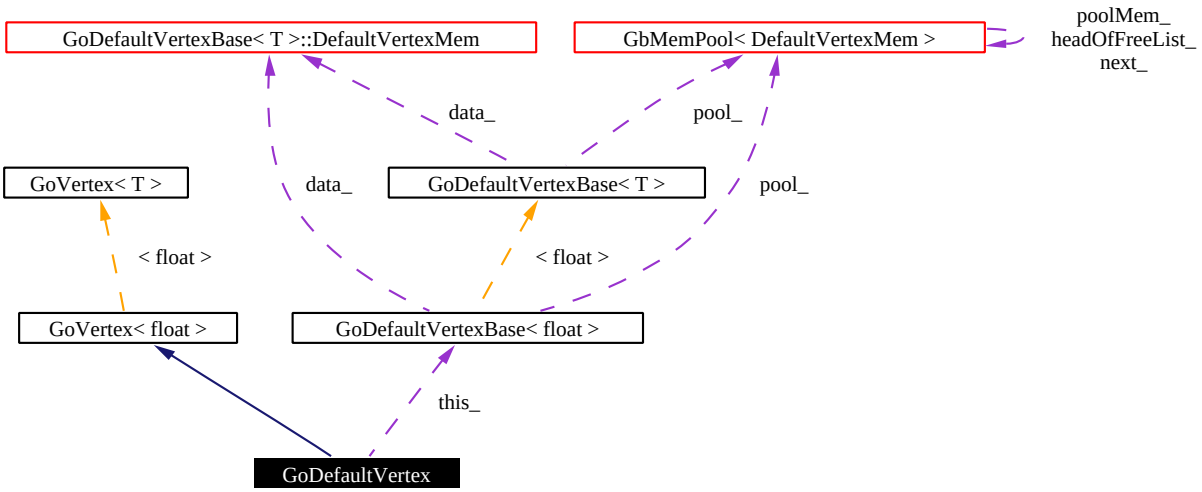


Figure C.5: Collaboration graph for *gridlib* vertices.

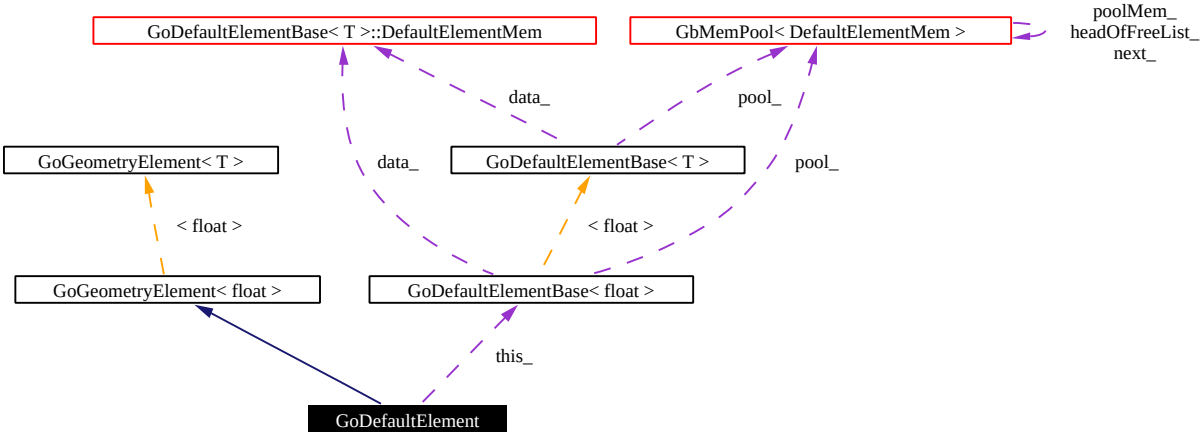


Figure C.6: Collaboration graph for *gridlib* geometry elements.

D

THE *gridlib* MESH INTERFACE

The mesh abstraction layer provides the most useful interface to clients. Here is a list of methods and descriptions of the supported operations. The mesh interface is implemented in the `GoMesh` class. See the source code documentation for details.

Task	*Functionality*
Geometry element handling	Adding and retrieving geometry elements (global ordering). Get the number of contained geometry elements. All methods offer constant time complexity.
Edge handling	Adding and retrieving directed edges (global ordering). Get the number of contained edges. All methods offer constant time complexity.
Vertex handling	Adding and retrieving vertices (global ordering). Get the number of contained vertices. Get the geometry element or edge associated with a vertex. Iterate over local vertices of a geometry element. All methods offer constant time complexity.
Adjacency information	Retrieve a list of geometry elements that surround an edge or vertex. Get all edges emanating from or incident to a vertex. Get a list of vertices related to a specific vertex by an edge. All methods can be parameterized to consider only items that have a specific flag combination set. All methods operate on local neighbor information and therefore offer constant time complexity.
Spacial extent	Get the axis-aligned bounding box of the mesh. Get an oriented bounding box containing all vertices or only vertices marked by a flag combination. All methods offer linear time complexity.
Content information	Querying the type of geometry elements in the container. Can be specific for each element type or as general 2D/3D statement in constant time complexity. Removal of unreferenced items in linear time complexity.

Algorithmic support	Methods for executing a given functor on all contained geometry elements, edges or vertices. Can be restricted to a specific subdivision level. Computing of surface or vertex normals, reconstructing neighbor information and creating edges from scratch. Consistent global renumbering of all items in user-provided identifier field. All methods offer linear time complexity.
Subdivision	Get the number of subdivision levels. Perform adaptive subdivision according to a provided oracle.

Bibliography

[ACG97] Rob Appelbaum, Marshall Cline, and Mike Girou. *The CORBA FAQ*, 1997. `http://www.cerfnet.com/~mpcline/corba-faq/`.

[Amd67] G.M. Amdahl. Validity of the single–processor approach to achieving large scale computation capabilities. *AFIPS Conference Proceedings*, 30:483–485, 1967.

[AMD00] Advanced Micro Devices (AMD). *3DNow! Technology Manual*, 2000. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/21928.pdf`.

[AMD02] Advanced Micro Devices (AMD). *AMD Athlon Processor x86 Code Optimization Guide*, 2002. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf`.

[ART99] The AR250 - a new architecture for ray traced rendering. In *Eurographics/SIGGRAPH workshop on graphics hardware - Hot topics session*, pages 39–42. Advanced Rendering Techniques, 1999.

[Ben75] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BK93] H.E. Bal and M.F. Kaashoek. Object distribution in Orca using compile–time and run–time techniques. *OOPSLA '93 Conference Proceedings*, 28(10):162–177, october 1993.

[Cel01] Celoxica Limited. *The Handel-C programming language*, 2001. `http://www.celoxica.com/products/technical_papers/datasheets/DATHNC002_0.pdf`.

[CL93] B. Cabral and L. Leedom. Imaging Vector Fields Using Line Integral Convolution. In J. T. Kajiya, editor, *Computer Graphics Proceedings*, volume 27 of *Annual Conference Series*, pages 263–270, Los Angeles, California, August 1993. ACM SIGGRAPH, Addison-Wesley Publishing Company, Inc.

[CR98] A. G. Chalmers and E. Reinhard. Parallel and distributed photo-realistic rendering. *SIGGRAPH Course Notes - Course 3*, july 1998.

[Cro98] Thomas W. Crockett. Parallel rendering. In *SIGGRAPH '98 "Parallel Graphics and Visualization Technology" course #42 notes*, pages 157–207. ACM, July 1998.

[CS98] Chris Cleeland and Douglas C. Schmidt. External Polymorphism — An Object Structural Pattern for Transparently Extending C++ Concrete Data Types. *C++ Report Magazine*, September 1998. `http://www.cs.wustl.edu/~schmidt/C++-EP.ps.gz`.

[DHH+00a] C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Wei. Portable memory hierarchy techniques for PDE solvers: Part I. *SIAM News*, 33(5), june 2000.

[DHH+00b] C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Wei. Portable memory hierarchy techniques for PDE solvers: Part II. *SIAM News*, 33(6), july 2000.

[FK97] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9), september 1972.

[Frü94] Thomas Frühauf. Interactive visualization of vector data in unstructured volumes. *Computers and Graphics*, 18:73–80, 1994.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, first edition, 1995.

[Gil93] W. K. Giloi. *Rechnerarchitektur*. Springer Verlag, second edition, 1993.

[GKLT00] G. Greiner, P. Kipfer, U. Labsik, and U. Tremel. An object oriented approach for high performance simulation and visualization on adaptive hybrid grids. In *Proceedings SIAM CSE Conference, Washington*, 2000.

[GLFK98] Andrew S. Grimshaw, Michael J. Lewis, Adam J. Ferrari, and John F. Karpovich. Architectural support for extensibility and autonomy in wide-area distributed object systems. Technical Report CS-98-12, University of Virginia, june 1998.

[HA97] Alan Heirich and James Arvo. Parallel rendering with an actor model. *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics*, pages 115–125, september 1997.

[HKRG02] F. Hülsemann, P. Kipfer, U. Rüde, and G. Greiner. *gridlib*: Flexible and efficient grid management for simulation and visualization. In Peter M. A. Sloot, C. J. Kenneth Tan, Jack J. Dongarra, and Alfons G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2331, pages 652–661. Springer, Berlin, 2002.

[Int00] Intel Corporation. *Getting Started with SSE/SSE2 for the Intel Pentium 4 Processor*, 2000. `http://cedar.intel.com/media/pdf/p4/getting_started.pdf`.

[Int02] Intel Corporation. *Hyper-Threading Technology Implications and Setup on Linux Operating Systems*, 2002. `http://cedar.intel.com/cgi-bin/ids.dll/content/content.jsp?cntKey=Generic+Editorial%3a%3asolveit_impLinuxSetup&cntType=IDS_EDITORIAL&catCode=CDN`.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In Xavier Pueyo and Peter Schröder, editors, *Rendering Techniques '96 (Proceedings Seventh Eurographics Workshop on Rendering)*, pages 21–30. Springer, June 1996.

[Jez93] J.-M. Jezequel. EPEE: an Eiffel environment to program distributed memory parallel computers. *Journal of Object Oriented Programming*, 6(2):48–54, May 1993.

[KG01] P. Kipfer and G. Greiner. Parallel rendering within the integrated simulation and visualization framework "*gridlib*". *VMV Conference Proceedings, Stuttgart*, 2001.

[KHM+02] P. Kipfer, F. Hülsemann, S. Meinlschmidt, B. Bergen, G. Greiner, and U. Rüde. *gridlib*: A parallel, object-oriented framework for hierarchical-hybrid grid structures in technical simulation and scientific visualization. In A. Bode, F. Durst, W. Hanke, and S. Wagner, editors, *High Performance Computing in Science and Engineering 2000 - 2002*, pages 489–501. Springer, Munich, 2002. to appear.

[Kip00] Peter Kipfer. *gridlib*: System design. Technical Report 4/00, Computer Graphics Group, University of Erlangen-Nürnberg, 2000.

[Kip01a] Peter Kipfer. *gridlib*: Advanced object-oriented programming paradigms. Technical Report 4/01, Computer Graphics Group, University of Erlangen-Nürnberg, 2001.

[Kip01b] Peter Kipfer. *gridlib*: Numerical methods. Technical Report 2/01, Computer Graphics Group, University of Erlangen-Nürnberg, 2001.

[Kla89] Rainer Klar. *Digitale Rechenautomaten*. de Gruyter, fourth edition, 1989.

[Kni96] G. Knittel. A PCI-compatible FPGA-processor for 2D/3D image processing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 136–145. IEEE, april 1996.

[KON] KONWIHR. *Competence Network for Technical, Scientific High Performance Computing in Bavaria*. `http://konwihr.in.tum.de/index_e.html`.

[KRG02] P. Kipfer, F. Reck, and G. Greiner. Local exact particle tracing on unstructured grids. *Computer Graphics Forum*, 22(2), 2002. accepted for publication.

[KS99] Peter Kipfer and Philipp Slusallek. Transparent distributed processing for rendering. *Proceedings Parallel Visualization and Graphics Symposium (PVG)*, pages 39–46, 1999.

[KSK97] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. Technical report, Department of Computer Science and Engineering, University of Minnesota, 1997.

[Lar98] Greg Ward Larson. The holodeck: A parallel ray-caching rendering system. In *Eurographics Workshop on Parallel Graphics and Visualization*, 1998.

[LB98] Adriano Lopes and Ken Brodlie. Accuracy in 3D particle tracing. In Hans-Christian Hege and Konrad Polthier, editors, *Mathematical Visualization*, pages 329–341. Springer Verlag, Heidelberg, 1998.

[LKG00] U. Labsik, P. Kipfer, and G. Greiner. Visualizing the structure and quality properties of tetrahedral meshes. Technical Report 2/00, Computer Graphics Group, University of Erlangen-Nürnberg, 2000.

[LKMG01] U. Labsik, P. Kipfer, S. Meinlschmidt, and G. Greiner. Progressive isosurface extraction from tetrahedral meshes. In H. Suzuki, A. Rockwood, and L. Kobbelt, editors, *Proceedings of Pacific Graphics '01 Conference, Tokyo*, pages 244–253, 2001.

[LL00] P. Lallemand and L. Luo. Theory of the Lattice-Boltzmann method: Dispersion, dissipation, isotropy, galilean invariance and stability. *Physical Review E*, 61, 2000.

[LS96] R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In James O. Coplien, John Vlissides, and Norm Kerth, editors, *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, MA, 1996. `http://www.cs.wustl.edu/~schmidt/Active-Objects.ps.gz`.

[MA] Jane F. Macfarlane and Rob Armstrong. *POET: A parallel object-oriented environment and toolkit for enabling high-performance scientific computing*. `http://omega.lbl.gov/poet/Poet.html`.

[McK95] Paul E. McKenney. Selecting locking primitives for parallel programs. Technical report, Sequent Computer Systems, Inc., 1995. `http://c2.com/ppr/mutex/mutexpat.html`.

[Mer99] Philippe Merle. The CorbaScript language. Technical report, Université des Sciences et Technologies de Lille, 1999. `http://corbaweb.lifl.fr/CorbaScript/`.

[ML95] Michael J. Muuss and Maximo Lorenzo. High-resolution interactive multispectral missile sensor simulation for ATR and DIS. In *BRL-CAD Symposium*, 1995.

[Mot99] Motorola Inc. *AltiVec Technology Programming Interface Manual*, 1999. `http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPIM.pdf`.

[MPI97] *The Message Passing Library Version 2.0*, 1997. `http://www.mpi-forum.org`.

[MPJ+00] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *IEEE International Symposium on Computer Architecture*, 2000.

[NJ99] Gregory M. Nielson and II-Hong Jung. Tools for computing tangent curves for lineary varying vector fields over tetrahedral domains. *IEEE Transactions on Visualization and Computer Graphics*, pages 360–372, 1999.

[NJS+97] G. M. Nielson, I.-H. Jung, N. Srinivasan, J. Sung, and J.-B. Yoon. Tools for Computing Tangent Curves and Topological Graphs for Visualizing Piecewise Linearly Varying Vector Fields over Triangulated Domains. In G. M. Nielson, H. Hagen, and H. Müller, editors, *Scientific Visualization: Overviews, Methodologies, and Techniques*, chapter 21, pages 527–562. IEEE Computer Society Press, Los Alamitos, California, 1997.

[NL01] Nallatech Limited. *IEEE 754 Floating Point Core*, 2001. `http://www.nallatech.com`.

[NVa] nVIDIA Corporation. *New NVIDIA GPU Breaks One Billion Pixels Per Second Barrier*. `http://www.nvidia.com/view.asp?IO=IO_20010618_6339`.

[NVb] nVIDIA Corporation. *Texture Compositing with Register Combiners*. `http://developer.nvidia.com/docs/IO/1382/ATT/RegisterCombiners.pdf`.

[NVc] nVIDIA Corporation. *Vertex shaders*. `http://www.nvidia.com/docs/IO/92/ATT/vertexshaders.pdf`.

[OG02] Carl Ollivier-Gooch. *GRUMMP Version 2.0.0 User's Guide*, 2002. `http://tetra.mech.ubc.ca/GRUMMP`.

[OMG95a] Object Management Group. *Compound Presentation and Compound Interchange Facilities, Part I*, December 1995. OMG Document 95-12-30.

[OMG95b] Object Management Group. *CORBAServices: Common Object Services Specification*, March 1995. updated November 22, 1996.

[OMG95c] Object Management Group. *Systems Management: CommonManagement Facilities, Volume 1, Version 2*, December 1995. OMG Documents 95-12-02 through 95-12-06.

[OMG96] Object Management Group. *Description of New OMA Reference Model, Draft 1*, May 1996. OMG Document 96-05-02.

[OMG97a] Object Management Group. *OMG Home Page*, 1997. `http://www.omg.org/`.

[OMG97b] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.1 edition, 1997. OMG Document 97-09-01.

[OMP02] OpenMP Architecture Review Board. *OpenMP Specification for C++, Version 2.0*, March 2002. `http://www.openmp.org/specs/mp-documents/cspec20.pdf`.

[PBMH02] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *Proceedings of SIGGRAPH 2002 Conference*, pages 703–712, 2002.

[PHK+99] Hans-Peter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro realtime ray casting system. In *Computer Graphics, 33 (Annual Conference Series)*, pages 251–260, 1999.

[PK01] Hans-Peter Pfister and K. Kreeger. RAYA: A ray tracing architecture for volumes and polygons. In *SIGGRAPH 2001 course on interactive ray tracing*, 2001. `http://www.merl.com/papers/TR99-19/`.

[PMS+99] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. *Interactive 3D Graphics (I3D)*, pages 119–126, 1999.

[PS98] Irfan Pyarali and Douglas C. Schmidt. An Overview of the CORBA Portable Object Adapter. *ACM StandardView magazine on CORBA*, 1998. `http://www.cs.wustl.edu/~schmidt/POA.ps.gz`.

[RCJ98] E. Reinhard, A. G. Chalmers, and F. W. Jansen. Overview of parallel photo-realistic graphics. *EUROGRAPHICS'98 - State-of-the-Art Reports*, pages 1–25, august 1998.

[RSHTE99] C. Rezk-Salama, P. Hastreiter, C. Teitzel, and T. Ertl. Interactive animation of volume line integral convolution based on 3D texture mapping. In *Visualization '99 Conference Proceedings*, Vienna, Austria, 1999. IEEE Computer Society Press.

[SB94] S. Singh and P. Bellec. Virtual hardware for graphics applications using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 49–58. IEEE Computer Society Press, 1994.

[Sch94] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the 12th Annual Sun Users Group Conference*, pages 214–225, San Francisco, CA, June 1994. SUG. `http://www.cs.wustl.edu/~schmidt/SUG-94.ps.gz`.

[Sch01a] Michael Schröder. *Automatische Objekt- und Threadverteilung in einer virtuellen Maschine*. PhD thesis, Institut für Informatik, Universität Erlangen-Nürnberg, 2001.

[Sch01b] M. Schrumpf. Beschleunigte Isoflächenberechnung auf unstrukturierten Gittern. *Studienarbeit*, 2001. Computer Graphics Group, University of Erlangen-Nürnberg.

[SGT96] D.J. Scales, K. Gharacharloo, and C.A. Thekkath. Shasta: A low–overhead, software–only approach for fine–grain shared memory. *Proceedings of the 7th symposium on architectural support for programming languages and operating systems*, october 1996.

[SHP97] Douglas C. Schmidt, Timothy H. Harrison, and Nat Pryce. Thread-specific storage for C/C++. In *Pattern Languages of Programming '97 conference proceedings*, September 1997. `http://www.cs.wustl.edu/~schmidt/TSS-pattern.ps.gz`.

[SHSS00] Marc Stamminger, Jörg Haber, Hartmut Schirmacher, and Hans-Peter Seidel. Walkthroughs with corrective textures. In *11th Eurographics workshop on rendering*, pages 377–388, 2000.

[SL00] Henry Styles and Wayne Luk. Customising graphics applications: Techniques and programming interface. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2000.

[Slu96] Philipp Slusallek. *Vision — an Architecture for Physically-Based Rendering*. PhD thesis, Computer Graphics Group, University of Erlangen-Nürnberg, Germany, 1996.

[Sol95] Richard Mark Soley, editor. *Object Management Architecture Guide*. John Wiley & Sons, third edition, 1995.

[SSH+98] Philipp Slusallek, Marc Stamminger, Wolfgang Heidrich, Jan-Christian Popp, and Hans-Peter Seidel. Composite lighting simulations with lighting networks. *IEEE Computer Graphics and Applications*, 18(2), March/April 1998.

[SSS98] Philipp Slusallek, Marc Stamminger, and Hans-Peter Seidel. Lighting networks – a new approach for designing lighting algorithms. *Graphics Interface*, pages 17–25, june 1998.

[SV96] Douglas C. Schmidt and Steve Vinoski. Distributed callbacks and decoupled communication in CORBA. *SIGS C++ Report*, October 1996.
`http://www.cs.wustl.edu/~schmidt/C++-report-col8.ps.gz`.

[TAO97] Computer Science Department, Washington University at St. Louis. *Real-time CORBA with TAO (The ACE ORB)*, 1997. `http://www.cs.wustl.edu/~schmidt/TAO.html`.

[TAO98] Computer Science Department, Washington University at St. Louis. *Principles and Patterns of High-performance, Real-time Object Request Brokers*, 1998.
`http://www.cs.wustl.edu/~schmidt/TAO4.ps.gz`.

[USM96] S. Ueng, K. Sikorski, and K. Ma. Efficient streamline, streamribbon, and streamtube constructions on unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 2:100–110, 1996.

[Wal95] Klaus Waldschmidt, editor. *Parallelrechner: Architekturen—Systeme—Werkzeuge*. Teubner, 1995.

[WG00] D. A. Wolf-Gladrow. *Lattice Gas Cellular Automata and Lattice Boltzmann Models*. Springer, 2000.

[WH92] Gregory J. Ward and Paul Heckbert. Irradiance gradients. *Third Eurographics Workshop on Rendering*, pages 85–98, May 1992.

[WSB01] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive distributed ray tracing of highly complex models. In *Eurographics workshop on rendering*, pages 274–285. Rendering Techniques 2001, 2001.

[WSBW01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. *Eurographics 2001 Proceedings*, 20(3), 2001.

[Xil01a] Xilinx. *Virtex datasheet*, 2001.
`http://www.xilinx.com/partinfo/ds031.htm`.

[Xil01b] Xilinx. *Virtex II Platform FPGA Handbook*, 2001.
`http://www.xilinx.com/products/virtex/handbook/ug002.pdf`.

Index

Is there another word for synonym ?

— *anonym*

gridlib, 63

abstraction layers, 64
ACE, 30
active de-multiplexing, 37
Active Object Table, 37
ADAPTIVE Communication Environment, *see* ACE
algorithmic abstraction, 70
Application Interface, 34
application specific integrated circuits, *see* ASICs
ASICs, 55

Basic Object Adapter, *see* BOA
BOA, 34, 36

C++ Wrapper, 31
cache, 21, 49
cache coherent NUMA, *see* ccNUMA
ccNUMA, 22
CFD, 97
CLB, 55
client, 32
Common Facilities, 34
Common Object Request Broker Architecture, *see* CORBA
Common Object Services Specification, *see* COSS
communicator, 28
computational fluid dynamics, *see* CFD
concurrent parallel, 22
configurable logic blocks, *see* CLB
Connection Strategy, 35
container, *see* mesh
CORBA, 32
COSS, 34
custom programmable hardware, 54

data parallel, 22
de-multiplexing, 34
derived MPI data type, 29
design patterns, 31
DII, 33
distributed shared memory, *see* DSM
Domain Interface, 34
DSI, 33
DSM, 41
dynamic dispatch, 34
Dynamic Invocation Interface, *see* DII
Dynamic Skeleton Interface, *see* DSI

ensemble averaging, 84
exact integration method, 98
external polymorphism, 67, 69

facilities, 32
false sharing effect, 41
field programmable gate arrays, *see* FPGA
floating point operations per second, *see* FLOPS
FLOPS, 49
FPGA, 54
framework, 31
full-custom design, 55
functional decomposition, 42
functional parallel, 22
functor, 70

General Inter–ORB Protocol, *see* GIOP
GIOP, 34
grand challenge, 21

IDL, 33
IIOP, 34
Inter–ORB Protocol, 34
Interface Definition Language, *see* IDL
Interface Repository, 33
Internet Inter–ORB Protocol, *see* IIOP
Interoperable Object References, *see* IOR
IOR, 34

Language Mapping, 33
Lattice Boltzmann, 84
lattice gas, 83
lattice site updates per second, *see* LUPS
LIC, 91
line integral convolution, *see* LIC
LUPS, 87

marching cubes, 78
marshaling, 33
memory pool, 65
mesh, 70
message passing, 26
Message Passing Library, *see* MPI
middleware, 63
MIMD, 21
MISD, 21
MPI, 27
multiple instruction, multiple data, *see* MIMD
multiple instruction, single data, *see* MISD

no remote memory access, *see* NORMA
non uniform memory access, *see* NUMA
NORMA, 21
NUMA, 21
numerical simulations, 63

Object Adapter, 34, 36
Object Management Architecture, *see* OMA
Object Management Group, *see* OMG
Object model, 32
Object Request Broker, *see* ORB
OMA, 32
OMG, 32
OpenMP, 43
ORB, 32
ORB Core, 32
OS Adaption Layer, 31

Parallel Virtual Machine, *see* PVM
particle distribution function, 84
personalization, 55
pipelining, 42
POA, 34, 36
POA Manager, 37
Portable Object Adapter, *see* POA
primitive objects, 69
programmable hardware, 53
proxy interfaces, 67
PVM, 26

QoS, 34
Quality–of–Service, *see* QoS

real–time, 34
Reference model, 32
register combiners, 54
relaxation, 84
request, 32
request-callback scheme, 106
RootPOA, 37

semi-custom design, 55
server, 32
services, 32
shear-warp algorithm, 92
SIMD, 21
single instruction, multiple data, *see* SIMD
single instruction, single data, *see* SISD
single program, multiple data, *see* SPMD
SISD, 21
Skeleton, 33
skeleton program, 70
SPMD, 22
Stub, 33
supercomputers, 21

TAO, 34
The ACE ORB, *see* TAO
thread–pool, 36

UMA, 21
uniform memory access, *see* UMA

vertex shader, 54
virtual NUMA, 41