Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures

Datenlokalitätsoptimierungen für iterative numerische Algorithmen und zelluläre Automaten auf Architekturen mit hierarchischem Speicher

Der Technischen Fakultät der

Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR–INGENIEUR

vorgelegt von

Dipl.-Inf. Markus Kowarschik

Erlangen — 2004

Als Dissertation genehmigt von

der Technischen Fakultät der

Universität Erlangen-Nürnberg

Tag der Einreichung:	16. April 2004
Tag der Promotion:	18. Juni 2004
Dekan:	Prof. Dr. A. Winnacker
Berichterstatter:	Prof. Dr. U. Rüde
	Prof. Dr. A. Bode

Abstract

In order to mitigate the impact of the constantly widening gap between processor speed and main memory performance on the runtimes of application codes, today's computer architectures commonly employ hierarchical memory designs including several levels of cache memories. Efficient program execution can only be expected if the underlying hierarchical memory architecture is respected. This is particularly true for numerically intensive codes.

Unfortunately, current compilers are unable to introduce sophisticated cache-based code transformations. As a consequence, much of the tedious and error-prone optimization effort is left to the software developer. In the case of a parallel numerical application running on a cluster of workstations, for example, hierarchical memory optimization represents a crucial task that must be addressed in addition to typical parallelization objectives such as efficient load balancing and the minimization of communication overhead.

The tuning of the utilization of the memory hierarchy covers a wide spectrum of techniques ranging from hardware-based technologies (e.g., data prefetching mechanisms) to compiler-based code transformations (e.g., restructuring loops) as well as fundamental algorithmic modifications. The latter may treat increased computational work for reduced memory costs, for example, and are a long way from being introduced automatically by a compiler. In general, achieving both high runtime efficiency and code portability represents a software engineering challenge that must be faced whenever designing numerical libraries.

In this thesis, we will present approaches towards the optimization of the data locality of implementations of grid-based numerical algorithms. In particular, we will concentrate on multigrid methods based on structured meshes as well as cellular automata in both 2D and 3D. As a representative example of cellular automata, we will consider the lattice Boltzmann method for simulating fluid flow problems.

Furthermore, we will discuss a novel approach towards inherently cache-aware multigrid methods. Our algorithm, which is based on the theory of the fully adaptive multigrid method, employs a patch-adaptive processing strategy. It is thus characterized by a high utilization of the cache hierarchy. Moreover, due to its adaptive update strategy, this scheme performs more robustly than classical multigrid algorithms in the case of singularly perturbed problems.

Acknowledgments

I would like to thank my advisor Prof. Dr. Ulrich Rüde (LSS, University of Erlangen-Nuremberg) for pointing me to the challenging research area of locality optimizations and for his support during my time as a graduate student in his research group. Thank you for your friendly assistance and your constant willingness to explain and discuss complex issues in both computer science and mathematics! I also wish to thank Prof. Dr. Arndt Bode (LRR, TU Munich) for reviewing this thesis.

I wish to give very special thanks to all my current and former colleagues and friends of the LSS team. Thank you very much for your cooperativeness which I have deeply appreciated! I have enjoyed our numerous and valuable scientific discussions throughout the years. Many LSS team members did a great job in proofreading large parts of this thesis.

Additionally, I would like to thank both my former colleague Dr. Christian Weiß and my current colleague Dr. Josef Weidendorfer (LRR, TU Munich) for the friendly and productive collaboration in our DiME project.

In particular, I would like to strongly emphasize and to express my gratitude to all the students with whom I had the pleasure to collaborate over the last few years: Iris Christadler, Volker Daum, Klaus Iglberger, Harald Pfänder, Nils Thürey, Harald Wörndl-Aichriedler, Jens Wilke, and Markus Zetlmeisl contributed many important pieces to my research that has finally led to the writing of this thesis.

I am further indebted to Dr. Daniel Quinlan from the Center for Applied Scientific Computing at the Lawrence Livermore National Laboratory for the great opportunity to spend the summers of 2001 and 2002 in an exciting working environment at the Livermore Lab where I could contribute to the research project ROSE. Moreover, I owe thanks to Prof. Dr. Craig C. Douglas from the Center for Computational Sciences at the University of Kentucky. Our intensive joint research efforts have always had a significant impact on my work.

Last but not least, I would like to express my deep gratitude to my girlfriend Ela as well as my dear parents. Thanks for your patience and for your invaluable support!

Markus Kowarschik

Contents

1 Introduction **1**

1.1 Scope . . . 1

1.2 The *Big Picture* — Motivation and Contributions . . . 1

1.2.1 Performance Tuning of Numerical Applications . . . 1

1.2.2 Optimization Targets . . . 2

1.2.3 Optimization Approaches . . . 3

1.2.4 The DiME Project . . . 3

1.3 Outline . . . 4

I Memory Hierarchies and Numerical Algorithms **5**

2 Memory Hierarchies **7**

2.1 Overview: Improving CPU Performance . . . 7

2.2 Organization of Memory Hierarchies . . . 7

2.2.1 Motivation . . . 7

2.2.2 CPU Registers, Cache Memories, and Main Memory . . . 8

2.2.3 Translation Lookaside Buffers . . . 10

2.3 The Principle of Locality . . . 10

2.4 Characteristics of Cache Architectures . . . 11

2.4.1 Cache Lines and Set Associativity . . . 11

2.4.2 Replacement Policies . . . 11

2.4.3 Write Policies . . . 12

2.4.4 Classification of Cache Misses . . . 13

2.5 Cache Performance Analysis . . . 13

2.5.1 Code Profiling — Techniques and Tools . . . 13

2.5.1.1 Hardware Performance Counters . . . 13

2.5.1.2 Instrumentation . . . 14

2.5.2 Simulation-Based Cache Performance Analysis . . . 15

3 Numerical Algorithms I — Iterative Linear Solvers **17**

3.1 Preparation: Discretization of PDEs . . . 17

3.2 Elementary Linear Solvers . . . 18

3.2.1 Overview: Direct Methods vs. Iterative Methods . . . 18

3.2.1.1 Preparations . . . 18

3.2.1.2 Direct Methods . . . 19

3.2.1.3 Iterative Methods . . . 19
3.2.1.4 Why We Focus on Iterative Methods . . . 22
3.2.2 Jacobi's Method . . . 23
3.2.3 The Method of Gauss-Seidel . . . 23
3.2.4 Jacobi Over-Relaxation and Successive Over-Relaxation . . . 25
3.2.5 Remarks . . . 26
3.3 Multigrid Methods . . . 27
3.3.1 Overview . . . 27
3.3.2 Preparations . . . 28
3.3.3 The Residual Equation . . . 29
3.3.4 Convergence Behavior of Elementary Iterative Methods . . . 29
3.3.5 Aspects of Multigrid Methods . . . 30
3.3.5.1 Coarse-Grid Representation of the Residual Equation . . . 30
3.3.5.2 Inter-Grid Transfer Operators . . . 31
3.3.5.3 Coarse-Grid Operators . . . 33
3.3.5.4 Two-Grid Method . . . 33
3.3.5.5 Generalization to Multigrid . . . 34
3.3.5.6 Remarks on Multigrid Convergence Analysis . . . 37
3.3.5.7 Full Approximation Scheme . . . 37
3.3.5.8 Nested Iteration and Full Multigrid . . . 40

4 Numerical Algorithms II — Cellular Automata 43
4.1 Formal Specification of CA . . . 43
4.2 The Lattice Boltzmann Method . . . 45
4.2.1 Introduction . . . 45
4.2.2 Discretization of the Boltzmann Equation . . . 46
4.2.3 The LBM in 2D . . . 47
4.2.4 The LBM in 3D . . . 49
4.2.5 Boundary Conditions . . . 49
4.2.6 Abstract Formulation of the LBM . . . 51
4.2.7 LBM Model Problem: Lid-Driven Cavity . . . 52

II Code Optimization Techniques for Memory Hierarchies 55

5 Data Layout Optimizations 57
5.1 Basics . . . 57
5.2 Array Padding . . . 57
5.2.1 Motivation and Principle . . . 57
5.2.2 Intra-Array Padding in 2D . . . 59
5.2.3 Intra-Array Padding in 3D . . . 59
5.2.3.1 The Standard Approach . . . 59
5.2.3.2 The Nonstandard Approach . . . 60
5.2.3.3 Example: Nonstandard Intra-Array Padding for Gauss-Seidel . . . 62
5.3 Cache-Optimized Data Layouts . . . 64
5.3.1 Array Merging . . . 64
5.3.2 Cache-Optimized Data Structures for Iterative Linear Solvers . . . 64

5.3.3 Cache-Optimized Data Structures for CA . . . 66
5.4 Grid Compression for CA and Jacobi-type Algorithms . . . 67

6 Data Access Optimizations **71**
6.1 Basics . . . 71
6.2 Loop Interchange . . . 72
6.2.1 Motivation and Principle . . . 72
6.2.2 Loop Interchange for Iterative Linear Solvers . . . 73
6.2.3 Loop Interchange for CA and Jacobi-type Methods . . . 73
6.3 Loop Fusion . . . 74
6.3.1 Motivation and Principle . . . 74
6.3.2 Application of Loop Fusion to Red-Black Gauss-Seidel . . . 74
6.3.3 Application of Loop Fusion to the LBM . . . 77
6.4 Loop Blocking . . . 77
6.4.1 Motivation and Principle . . . 77
6.4.2 Loop Blocking in 2D . . . 79
6.4.2.1 Blocked Red-Black Gauss-Seidel . . . 79
6.4.2.2 Blocking for CA and Jacobi-type Methods . . . 79
6.4.3 Loop Blocking in 3D . . . 83
6.4.3.1 Blocked Red-Black Gauss-Seidel . . . 83
6.4.3.2 Blocking for CA and Jacobi-type Methods . . . 92
6.5 Further Data Access Optimizations . . . 97
6.5.1 Data Prefetching . . . 97
6.5.2 Data Copying . . . 98

7 Experimental Results **101**
7.1 Initial Remarks . . . 101
7.1.1 Combining Efficiency and Flexibility . . . 101
7.1.2 Purpose of this Chapter . . . 102
7.2 Iterative Methods on 2D Grids . . . 102
7.3 Iterative Methods on 3D Grids . . . 102
7.3.1 Description of the Implementations . . . 102
7.3.2 Performance Results . . . 104
7.3.2.1 Analysis of the Standard Multigrid Implementation . . . 104
7.3.2.2 Comparison of Data Layouts . . . 107
7.3.2.3 Comparison of Loop Orders . . . 109
7.3.2.4 Multigrid Performance — General Case . . . 111
7.3.2.5 Multigrid Performance — Special Case: Constant Coefficients . . . 120
7.4 The LBM in 2D . . . 121
7.4.1 Description of the Implementations . . . 121
7.4.2 Performance Results . . . 123
7.5 The LBM in 3D . . . 128
7.5.1 Description of the Implementations . . . 128
7.5.2 Performance Results . . . 128
7.6 Summary of this Chaper . . . 132

III Designing Inherently Cache-Efficient Multigrid Algorithms 135

8 Patch-Adaptive Multigrid 137
8.1 Overview . . . 137
8.2 Adaptive Relaxation . . . 138
8.2.1 Overview . . . 138
8.2.2 Definitions and Elementary Results . . . 138
8.2.3 Algorithmic Description of the Adaptive Relaxation Method . . . 140
8.3 Adaptive Relaxation in Multigrid Methods . . . 141
8.3.1 Overview . . . 141
8.3.2 Preparations . . . 142
8.3.3 The Extended Linear System . . . 143
8.3.3.1 Nonunique Representation of Fine-Grid Vectors . . . 143
8.3.3.2 Derivation of the Extended Linear System . . . 144
8.3.3.3 Properties of the Matrix of the Extended Linear System . . . 145
8.3.4 Iterative Solution of the Extended Linear System . . . 146
8.4 Introducing Patch Adaptivity . . . 146
8.4.1 Overview of the Patch-Adaptive Multigrid Scheme . . . 146
8.4.2 Patch Design . . . 148
8.4.3 Immediate Consequences of the Patch-Adaptive Approach . . . 150
8.4.3.1 Reduction of Granularity and Overhead . . . 150
8.4.3.2 Increased Update Rate . . . 150
8.4.4 Patch Processing . . . 151
8.4.4.1 Definition of the Scaled Patch Residual . . . 151
8.4.4.2 Patch Relaxation . . . 151
8.4.5 Activation of Neighboring Patches . . . 152
8.4.5.1 Problem Description . . . 152
8.4.5.2 Changes in Scaled Residuals . . . 152
8.4.5.3 Relaxation/Activation Policies . . . 152
8.4.6 Algorithmic Description of the Patch-Adaptive Relaxation Scheme . . . 154
8.4.7 Remarks on Implementation Details . . . 156
8.5 Experimental Results . . . 157
8.5.1 Description of the Model Problem . . . 158
8.5.1.1 Definition . . . 158
8.5.1.2 Analytical Solution . . . 158
8.5.1.3 Finite Element Discretization . . . 159
8.5.2 Numerical Tests . . . 161
8.5.2.1 Preparations . . . 161
8.5.2.2 Results . . . 164

IV Related Work, Conclusions, and Future Work 167

9 Related Work — Selected Topics 169
9.1 Optimization Techniques for Numerical Software . . . 169
9.1.1 Overview . . . 169
9.1.2 Automatic Performance Tuning of Numerical Software . . . 169

9.1.3 Further Optimization Techniques for Grid-Based Algorithms . . . 170
9.1.3.1 Cache Optimizations for Unstructured Grid Multigrid . . . 170
9.1.3.2 Hierarchical Hybrid Grids . . . 171
9.1.3.3 Cache-Efficient Parallel Domain Decomposition Approaches . . . 172
9.1.3.4 A Cache-Aware Parallel CA Implementation . . . 173
9.2 Cache Optimizations for the BLAS and LAPACK . . . 173
9.2.1 Overview of the BLAS and LAPACK . . . 173
9.2.2 Enhancing the Cache Performance of the BLAS . . . 173
9.2.3 Block Algorithms in LAPACK . . . 175
9.3 The DiMEPACK Library . . . 176
9.3.1 Functionality of the DiMEPACK Library . . . 176
9.3.2 Optimization Techniques . . . 177
9.3.2.1 Arithmetic Optimizations . . . 177
9.3.2.2 Locality Optimizations . . . 177
9.3.3 Performance of the DiMEPACK Library . . . 178
9.4 Examples of Locality Metrics . . . 178
9.4.1 Overview . . . 178
9.4.2 The Concept of Memtropy . . . 178
9.4.3 Definition and Application of a Temporal Locality Metric . . . 179
9.4.3.1 Motivation and Definition . . . 179
9.4.3.2 Example: Optimization of Lexicographic SOR in 2D . . . 180
9.5 Generation of Optimized Source Code Using ROSE . . . 183
9.5.1 Overview of ROSE . . . 183
9.5.2 The Infrastructure of ROSE . . . 183

10 Conclusions and Future Work **187**
10.1 Conclusions . . . 187
10.2 Practical Applications of our Work . . . 188
10.2.1 Bioelectric Field Simulations . . . 188
10.2.2 Chip Layout Optimization . . . 188
10.3 Suggestions for Future Work . . . 189
10.3.1 Integration of Cache Optimizations into Parallel Applications . . . 189
10.3.2 On-the-fly Assembly of Linear Equations . . . 189
10.3.3 Extension of the Patch-Adaptive Multigrid Scheme . . . 190

V Appendix **191**

A Machine Specifications **193**

B German Parts **195**
B.1 Inhaltsverzeichnis . . . 195
B.2 Zusammenfassung . . . 200
B.3 Einleitung . . . 201

C Curriculum Vitae **205**

List of Algorithms

3.1 Red-black Gauss-Seidel — basic structure. . . . 25
3.2 Two-grid CGC V(ν_1,ν_2)-cycle. . . . 35
3.3 Recursive definition of the multigrid CGC V(ν_1,ν_2)-cycle. . . . 36
3.4 Recursive definition of the multigrid FAS V(ν_1,ν_2)-cycle. . . . 39
3.5 Recursive formulation of the FMG scheme on Ω_h. . . . 40
4.1 Abstract formulation of the LBM, collide-and-stream update order. . . . 52
5.1 Inter-array padding. . . . 58
5.2 Intra-array padding in 2D. . . . 59
5.3 Standard intra-array padding in 3D. . . . 60
5.4 Nonstandard intra-array padding in 3D. . . . 61
5.5 Gauss-Seidel kernel without array padding in 3D. . . . 62
5.6 Gauss-Seidel kernel with nonstandard intra-array padding in 3D, Version 1. . . . 63
5.7 Gauss-Seidel kernel with nonstandard intra-array padding in 3D, Version 2. . . . 63
5.8 Array merging. . . . 64
6.1 Loop interchange. . . . 73
6.2 Loop fusion. . . . 75
6.3 Standard red-black Gauss-Seidel in 3D. . . . 75
6.4 Red-black Gauss-Seidel in 3D after loop fusion. . . . 76
6.5 Loop blocking for matrix transposition. . . . 78
6.6 Standard LBM in 2D. . . . 79
6.7 1-way blocking technique for the 2D LBM. . . . 80
6.8 3-way blocking technique for the 2D LBM. . . . 82
6.9 1-way blocked red-black Gauss-Seidel in 3D, Version 1. . . . 85
6.10 1-way blocked red-black Gauss-Seidel in 3D, Version 2. . . . 87
6.11 1-way blocked red-black Gauss-Seidel in 3D, Version 3. . . . 89
6.12 2-way blocked red-black Gauss-Seidel in 3D. . . . 90
6.13 3-way blocked red-black Gauss-Seidel in 3D. . . . 90
6.14 4-way blocked red-black Gauss-Seidel in 3D. . . . 91
6.15 4-way blocking technique for the 3D LBM. . . . 95
8.1 Sequential adaptive relaxation. . . . 140
8.2 Computing the scaled residual $\bar{r}_P$ of patch $P \in \mathcal{P}_l$. . . . 151
8.3 Relaxation of patch $P \in \mathcal{P}_l$. . . . 152
8.4 Patch-adaptive relaxation on grid level l. . . . 156

Chapter 1

Introduction

1.1 Scope

This work is intended to be a contribution to the field of research on high performance scientific computing. On the whole, this thesis focuses on architecture-driven performance optimizations for numerically intensive codes. In particular, it addresses the following areas:

- Memory hierarchy optimizations for numerically intensive codes
- Architecture-oriented development of efficient numerical algorithms
- Performance analysis and memory hierarchy profiling
- Development of flexible and efficient numerical software

1.2 The *Big Picture* — Motivation and Contributions

1.2.1 Performance Tuning of Numerical Applications

In order to increase the performance of any parallel numerical application, it is essential to address two related optimization issues, each of which requires intimate knowledge in both the algorithm and the architecture of the target platform. Firstly, it is necessary to minimize the parallelization overhead itself. These tuning efforts commonly cover the choice of appropriate parallel numerical algorithms and load balancing strategies as well as the minimization of communication overhead by reducing communication and hiding network latency and bandwidth. Much research has focused and will continue to be focused on the optimization of parallel efficiency.

Secondly, it is essential to exploit the individual parallel resources as efficiently as possible; i.e., by achieving the highest possible performance on each node in the parallel environment. This is especially true for distributed memory systems found in clusters based on off-the-shelf workstations communicating via fast interconnection networks. These machines commonly employ hierarchical memory architectures which are used in order to mitigate the effects of the constantly widening gap between CPU speed and main memory performance.

According to Moore's law from 1975, the number of transistors on a silicon chip will double every 12 to 24 months. This prediction has already proved remarkably accurate for almost three decades. It has led to an average increase in CPU speed of approximately 55% every year. In contrast, DRAM technology has evolved rather slowly. Main memory latency and memory bandwidth

have only been improving by about 5% and 10% per year, respectively [HP03]. The *International Technology Roadmap for Semiconductors*[1] predicts that this trend will continue further on and the gap between CPU speed and main memory performance will grow for more than another decade until technological limits will be reached. For a rich collection of contributions predicting and discussing future trends in micro- and nanoelectronics as well as their relation to Moore's law, we refer to [CiS03].

The research we will present in this thesis focuses on the aforementioned second optimization issue; the optimization of single-node efficiency. More precisely, we will describe and analyze techniques which primarily target the optimization of data cache performance on hierarchical memory architectures. In other words, the common aspect of all our approaches is that they exploit and optimize data locality. As a consequence, their application causes a reduction of both main memory accesses and cache misses, which directly translates into reduced execution times of numerical applications whose performance is primarily limited by memory latency or memory bandwidth.

Unfortunately, current optimizing compilers are not able to synthesize chains of complicated and problem-dependent cache-based code transformations. Therefore, they rarely deliver the performance expected by the users and much of the tedious and error-prone work concerning the tuning of single-node performance is thus left to the software developer. Only the proper combination of high parallel efficiency and optimized single-node performance leads to minimal application runtimes. This aspect sometimes seems to be neglected by the parallel computing community.

At this point, it is worth mentioning that typical loop bodies occurring in numerical applications are rather small and thus rarely exceed the capacity of today's instruction caches. As a consequence, the optimization of instruction cache performance does not represent a primary issue in high performance scientific computing. While additional research also addresses the optimization of accesses to external memory levels (e.g., hard disk drives), this is not considered a major practical research issue either [KW03].

1.2.2 Optimization Targets

In this thesis, we will target two algorithmically related classes of numerical algorithms; iterative linear solvers and cellular automata. These algorithms are characterized by large numbers of repetitive passes through the underlying data sets which can potentially be large.

Firstly, we will present and discuss locality optimization techniques for iterative solution methods for large sparse systems of linear equations. Such problems often arise in the context of the numerical solution of partial differential equations. We will address elementary iterative schemes first. However, it is not sufficient to target the optimization of algorithms which are comparatively inefficient from a mathematical point of view. Clearly, minimal solution times can only be expected from optimized implementations of optimal numerical algorithms.

Therefore, we will focus on the more advanced class of multigrid algorithms later on. These algorithms are typically based on elementary iterative schemes and can be shown to be asymptotically optimal. This means that, for many problems, the computational requirements of appropriately designed multigrid algorithms are comparatively low and grow only linearly with the number of unknowns of the linear system to be solved. The asymptotic optimality of multigrid methods illustrates their low computational intensity (i.e., the low ratio of floating-point operations per data access) and therefore emphasizes the particular urgent need for high data locality.

[1] See `http://public.itrs.net`.

Secondly, we will present and examine locality optimization approaches for cellular automata which are mainly employed in order to simulate and to study the time-dependent behavior of complex systems in science and engineering. We will use the lattice Boltzmann method as an example of cellular automata with high practical relevance, especially in the field of computational fluid dynamics as well as in material science research.

1.2.3 Optimization Approaches

On one hand, our work covers optimizations which could — at least theoretically — be introduced automatically by a compiler. These techniques aim at reordering the data layout as well as the data accesses such that enhanced reuse of cache contents results and larger fractions of data can be retrieved from the higher memory hierarchy levels. It is important to point out that these optimization approaches do not change any of the numerical properties of the algorithms under consideration. Rather, they aim at optimizing the execution speed of the application codes in terms of MFLOPS (i.e., millions of floating-point operations per second) in the case of iterative linear solvers and MLSUPS (i.e., millions of lattice site updates per second) in the case of the lattice Boltzmann method.

On the other hand, we will also address the design of inherently cache-aware numerical algorithms. As a matter of fact, the development of new efficient algorithms is a multifaceted field of ongoing research. In particular, we will introduce a novel multigrid scheme which is based on the theory of the fully adaptive multigrid method. This algorithm requires extensive mathematical analysis and does not result from the successive application of code transformations which may be automized. Our multigrid method represents a combination of patch-oriented mesh processing and adaptive relaxation techniques. As a consequence, it exhibits a high potential of data locality and can therefore be implemented very cache-efficiently.

In addition, due to the underlying adaptive relaxation approach, our patch-adaptive multigrid scheme follows the principle of local relaxation and thus performs more robustly than standard multigrid methods in the presence of operator-induced singularities or perturbations from boundary conditions. We will provide a detailed description of our algorithm and present experimental results.

1.2.4 The DiME Project

Our work has been funded by the *German Research Foundation (Deutsche Forschungsgemeinschaft, DFG)*, research grants Ru 422/7–1,2,3,5. It is partly based on the cooperation with Dr. Christian Weiß, a former staff member of the *Lehrstuhl für Rechnertechnik und Rechnerorganisation/Parallelrechnerarchitektur, Technische Universität München*, who finished his thesis in the end of 2001 [Wei01]. Both theses must necessarily be considered together in order to obtain a complete picture of the research that has been done and the variety of results that have been achieved. We will thus often refer to [Wei01] in the following.

We decided on the name

DiME — data-local iterative methods

for our joint project. This name has been kept and will be used further on although it does not encompass the additional research on locality optimizations for cellular automata.

1.3 Outline

This thesis is structured as follows. In Chapter 2, we will summarize those aspects of computer architecture which are relevant for our work. In particular, we will explain the design of memory hierarchies.

In Chapter 3, we will briefly introduce the first class of target algorithms that our optimization techniques will be applied to; iterative solution methods for large sparse systems of linear equations. We will present both elementary iterative solvers as well as multigrid algorithms.

Chapter 4 contains a description of the second class of target algorithms; cellular automata. We will provide the general formalism of cellular automata and then concentrate on the lattice Boltzmann method as a representative and practically relevant example.

In Chapters 5 and 6, we will present a variety of approaches in order to enhance the locality exhibited by the data layouts and the data access patterns. We will discuss the application of these techniques to both classes of algorithms from Chapters 3 and 4.

Experimental results will be presented in Chapter 7. They will cover both cache-optimized implementations of iterative linear solvers and the lattice Boltzmann method. We will address implementations of the algorithms in 2D and 3D.

Chapter 8 will focus on the design of inherently cache-aware multigrid algorithms for the numerical solution of partial differential equations. In particular, we will describe our patch-adaptive multigrid algorithm and present experimental numerical results.

An overview of selected topics of related work will be given in Chapter 9. We will present general performance optimization approaches for numerical software, including techniques for grid-based computations. We will discuss locality optimizations of algorithms of numerical linear algebra for dense matrices and introduce our cache-optimized multigrid library *DiMEPACK*. Afterwards, we will discuss various locality metrics and, as an example, examine the application of one of them to quantify the effect of a cache-based code transformation. Lastly, we will present an approach towards the automized introduction of library-specific source code transformations.

Final conclusions will be presented in Chapter 10. We will particularly include examples of practical applications of our work. Furthermore, we will suggest future research efforts in the context of architecture-oriented optimizations of numerical algorithms.

Part I

Memory Hierarchies and Numerical Algorithms

Chapter 2

Memory Hierarchies

2.1 Overview: Improving CPU Performance

According to [HP03], there are two fundamental approaches to improve the performance of a microprocessor; increasing the clock rate and reducing the average number of clock cycles per machine instruction. For these purposes, today's *scalar CPUs*[1] are characterized by many sophisticated hardware features which have in common that they all aim at minimizing the execution time of application codes. These hardware features include techniques to increase instruction-level parallelism (ILP) (e.g., pipelining, superscalarity), simultaneous multithreading (SMT), speculative execution and dynamic branch prediction to reduce the number of pipeline stalls, as well as hierarchical memory designs.

A discussion of each of these techniques would exceed the scope of this work. Instead, we refer to standard textbooks on computer architecture (e.g., [Bäh02a, Bäh02b, HP03]) and to the literature on numerical programming which covers efficiency aspects of scientific codes (e.g., [GH01, Ueb97a, Ueb97b]). Since we primarily focus on cache performance optimizations, we will restrict our explanations to aspects of hierarchical memory architectures in the remainder of this chapter. The following presentation is based on [Wei01] as well as on our joint publication [KW03].

2.2 Organization of Memory Hierarchies

2.2.1 Motivation

A *memory hierarchy* can be seen as an economic solution to the programmer's desire for a computer memory which is both large and fast [HP03]. The idea behind hierarchical memory designs is to hide both the relatively low main memory *bandwidth* as well as the relatively high *latency* of main memory accesses. In fact, main memory performance is generally very poor in contrast to the processing performance of the CPUs. This means that the speed with which current CPUs can process data greatly exceeds the speed with which the main memory can deliver data.

Only if the underlying hierarchical memory architecture is respected by the code, can efficient execution be expected. The impact of the constantly increasing gap between main memory performance and theoretically available CPU performance on the execution speed of a user's application is sometimes referred to as the *memory wall* [WM95].

[1] In contrast, the class of *vector CPUs* exhibits different architectural properties. These machines, however, are beyond the scope of this work. Instead, we refer to [HP03].

Generally, each *level* of a memory hierarchy contains a copy of the data stored in the next slower and larger level. The use of memory hierarchies is justified by the *principle of locality*, see Section 2.3. In the following, we will describe the structure of memory hierarchies.

2.2.2 CPU Registers, Cache Memories, and Main Memory

The top of the memory hierarchy usually consists of a small and expensive high speed memory layer, which is integrated within the processor in order to provide data with low latency and high bandwidth; i.e., the *CPU registers*. Moving further away from the CPU, the layers of memory successively become larger and slower. The memory hierarchy levels which are located between the processor core and main memory are called *cache memories (caches)*. These memory components are commonly based on fast semiconductor SRAM technology. They are intended to contain copies of main memory blocks to speed up accesses to frequently needed data [Han98, HP03][2].

Caches may be accessed using virtual (logical) addresses or physical (main memory) addresses. Correspondingly, if virtual addresses are used, the cache is said to be a *virtual (logical) cache*. Otherwise, it is said to be a *physical cache*. The major advantage of a virtual cache is that it can be searched immediately without waiting for the address translation to be completed by the delay-causing *memory management unit (MMU)*. On the other hand, physical caches do not have to be invalidated upon each context switch and they do not require the handling of *address aliasing* effects. Aliasing in the cache occurs if the operating system or user programs use different virtual addresses for the same physical memory location. It this case, it is necessary to implement nontrivial mechanisms (either in hardware or in software) to avoid that two or more copies of the same data item are kept in cache and modified independently of one another [Han98].

In order to combine the advantages of both approaches, hybrid forms such as *virtually indexed, physically tagged caches* have been developed and become popular in today's CPU designs [Bäh02a]. The idea behind this approach is to use a part of the page offset (which is identical for the virtual and the physical address) to index the cache immediately; i.e., to determine the corresponding cache addresses (i.e., the corresponding *set*, see below) to be searched. The physical address, which can be determined simultaneously by the MMU, is then used for the comparison of the tags to determine if the cache block to be read is actually in the cache. This hybrid approach combines the advantages of a virtual cache with the advantages of a physical cache.

In a typical memory hierarchy, a relatively small *first-level (L1) cache* is placed on the CPU chip to ensure low latency and high bandwidth. The L1 cache may be split into two separate parts; one for program data, the other one for instructions. The latency of on-chip L1 caches is often one or two clock cycles. In general, an on-chip cache runs at full CPU speed and its bandwidth scales linearly with the clock rate.

Chip designers face the problem that large on-chip caches of current high clock rate microprocessors cannot deliver data within such a small number of CPU cycles since the signal delays are too long. Therefore, the size of on-chip L1 caches is limited to 64 kB or even less for many of today's chip designs. However, larger L1 cache sizes with accordingly higher access latencies are becoming available.

An L1 cache is usually backed up by a *second-level (L2) cache*. A few years ago, architectures typically implemented the L2 cache on the motherboard (i.e., off-chip), using SRAM chip technology. However, today's L2 cache memories are commonly located on-chip as well; e.g., in the case of Intel's

[2]For an early overview of cache memories, we point to [Smi82]. Due to its time of publication, this article is particularly interesting from a historical point of view.

Itanium 2 CPU, see Figure 2.2. Currently, the capacity of on-chip L2 caches is increasing to more than 1 MB. They typically deliver data with a latency of approximately five to ten CPU cycles.

If the L2 cache is implemented on-chip as well, an on-chip or off-chip *third-level (L3) cache* may be added to the hierarchy. Today's on-chip L3 caches have capacities of up to 6 MB and provide data with a latency of about 10 to 20 CPU cycles (e.g., some Intel Itanium 2 CPUs, see again Figure 2.2). In contrast, current off-chip L3 cache sizes vary up to 32 MB. In general, off-chip caches provide data with significantly higher latencies and lower bandwidths than on-chip caches. For example, IBM's POWER4 systems are characterized by 32 MB off-chip L3 cache modules with relatively high latencies of up to 100 clock cycles [IBM01].

The next lower level of the memory hierarchy is the main memory which is large (typically several gigabytes) but also comparatively slow. Main memory components are commonly based on DRAM technology, for both economic and technical reasons. While SRAMs usually require six transistors to store a single bit, DRAMs use only a single one. This enables the development and the production of DRAM memory chips with relatively large capacities. However, DRAM technology requires the bits in memory to be refreshed periodically, which slows down the average memory access time [HP03].

Figure 2.1 is taken from [Wei01] and illustrates a typical hierarchical memory architecture with three levels of cache, where the L3 cache is located off-chip. Note that, for some processors, the assumption of cache inclusivity does not hold. In the case of the AMD Opteron and the AMD Athlon 64 CPUs, for example, there is no duplication of data between the L1 and the L2 cache. Instead, the L2 cache contains only data that has been evicted from the L1 cache [AMD03].

Figure 2.2 shows the memory architecture of a machine based on Intel's Itanium 2 CPU, which contains three levels of on-chip cache [Int03]. This architecture clearly indicates the current trend of growing on-chip caches. Note that a specific property of this CPU design is that the L1 cache does not store any floating-point numbers. Instead, floating-point loads bypass the L1 cache. We refer to Appendix A for cache characteristics of current architectures.

As Figure 2.2 further illustrates, external memory such as hard disk drives or remote memory in a distributed memory environment (e.g., a cluster of workstations) represent the lowest end of any common hierarchical memory architecture, unless even backup devices are considered as well.

Figure 2.2 reveals another crucial aspect. Comparing the bandwidths and latencies of the individual memory levels and the theoretically available performance of current CPUs, it becomes clear that moving data has become much more expensive than processing data. Consequently,

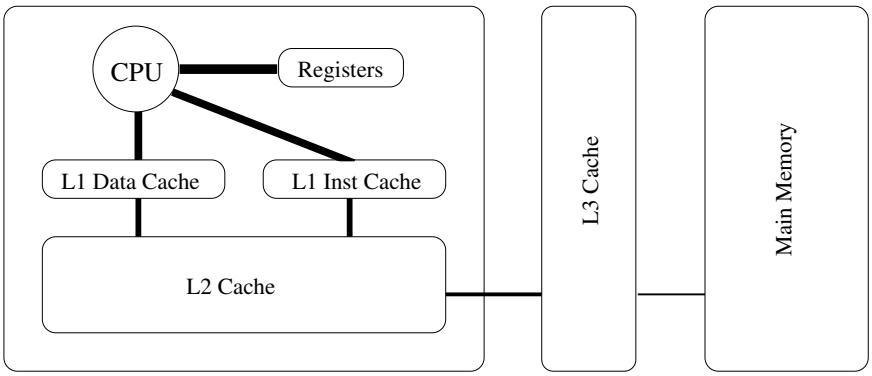


Figure 2.1: Hierarchical memory design.

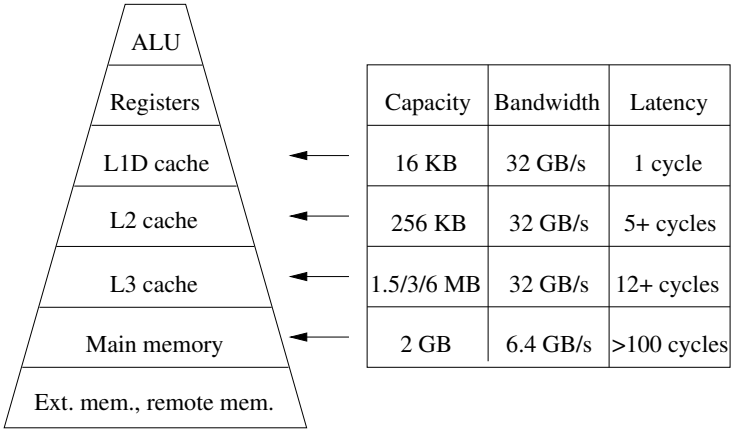


Figure 2.2: Memory hierarchy of Intel's Itanium 2 CPU running at a clock rate of 1 GHz and typical performance characteristics.

the number of floating-point operations alone can no longer be an adequate complexity measure. Rather, the locality behavior of the code and the corresponding costs of data accesses must be taken into consideration in order to obtain a reasonable measure of performance, cf. the presentation of *locality metrics* in Section 9.4.

2.2.3 Translation Lookaside Buffers

Finally, there is another type of cache memory which must be mentioned at this point. Virtual memory management requires the translation of virtual addresses into physical addresses. This address translation is commonly accomplished by the MMU, which is typically located on the CPU chip. In order to speed up this translation by avoiding accesses to the page table of the current process, which is stored in main memory, another type of cache is employed; the *translation lookaside buffer (TLB)*. Usually, a TLB is a fully associative cache (cf. Section 2.4.1) storing the physical addresses of virtual pages in main memory. It is often split into a data TLB and an instruction TLB. Depending on the actual design of the virtual memory system, TLBs can be organized hierarchically. Since they store neither instructions nor program data, they are not included in Figures 2.1 and 2.2.

Analogous to the caches for instructions and program data, the use of TLBs is motivated by the principle of locality, see Section 2.3. We refer to [Bäh02a, Han98, HP03] for technical details on TLBs. See also the machine specifications in Appendix A for typical TLB capacities and configurations.

2.3 The Principle of Locality

Because of their limited size, caches can only hold copies of recently used data or code. Typically, when new data is loaded into the cache, other data has to be replaced. Caches improve performance only if data which has already been loaded is reused before being replaced.

The reason why caches can substantially reduce program execution time is the principle of *locality of references* [HP03]. This principle is empirically established and states that most programs do not access all code or data uniformly. Instead, both recently used data as well as data stored nearby

the currently referenced data is very likely to be accessed in the near future.

Accordingly, locality can be subdivided into *temporal locality (locality by time)* and *spatial locality (locality by space)*. A sequence of references exhibits temporal locality if recently accessed data will be accessed again in the near future. A sequence of references exposes spatial locality if data located close together in address space tends to be referenced within a short period of time. Note that the code optimization techniques we will discuss in Chapters 5 and 6 aim at enhancing both temporal and spatial locality.

2.4 Characteristics of Cache Architectures

2.4.1 Cache Lines and Set Associativity

Data within the cache is stored in *cache frames (block frames)*. Each cache frame holds the contents of a contiguous block of memory; a so-called *cache line (cache block)*. The sizes of cache blocks vary from architecture to architecture. Typical values range from 4 B to 64 B. If the data requested by the processor can be retrieved from cache, it is called a *cache hit*. Otherwise, a *cache miss* occurs. The contents of the cache block containing the requested word are then fetched from a lower hierarchy layer and copied into a cache frame. For this purpose, another cache block must be replaced. Therefore, in order to guarantee low latency access, the question of into which cache frame the data should be loaded and how the data should be retrieved afterwards must be handled efficiently.

In respect to hardware complexity, the cheapest approach to implement block placement is *direct mapping*; the contents of a cache block can be placed into exactly one cache frame. Direct mapped caches have been among the most popular cache architectures in the past and are still common for off-chip caches.

However, computer architects have focused on increasing the *set associativity* of on-chip caches. An a-way set-associative cache is characterized by a higher hardware complexity, but usually implies higher hit rates. The cache frames of an a-way set-associative cache are grouped into sets of size a. The contents of any cache block can be placed into any cache frame of the corresponding set. Upon a memory request, an *associative search* is performed on the contents of the corresponding set in order to decide as quickly as possible if the required data is in cache or not, hence the increase in hardware complexity.

Finally, a cache is called *fully associative* if a cache block can be placed into any cache frame. Usually, fully associative caches are only implemented as small special-purpose caches; e.g., TLBs (see Section 2.2.3).

Figure 2.3 illustrates the positions where a cache block could possibly be placed in a fully associative, in a direct mapped, and in an a-way set-associative cache, respectively. Direct mapped and fully associative caches can be seen as special cases of a-way set-associative caches; a direct mapped cache is a 1-way set-associative cache, whereas a fully associative cache is C-way set-associative, provided that C is the total number of cache frames.

2.4.2 Replacement Policies

In a fully associative cache and in an a-way set-associative cache, a cache line can be placed into one of several cache frames. The question of into which cache frame a cache block is copied and

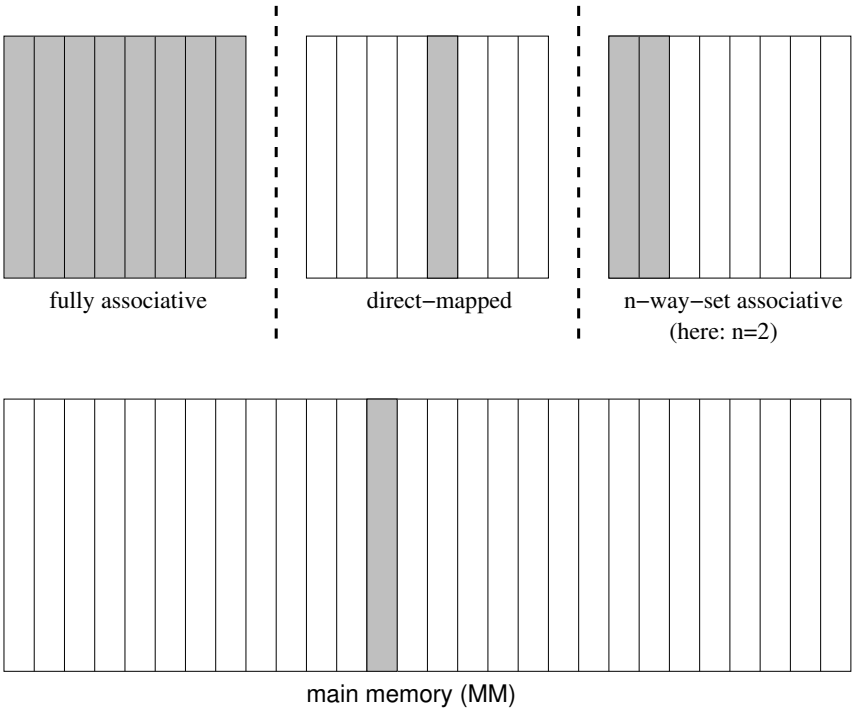


Figure 2.3: Degrees of set associativity of a cache memory.

which cache block thus has to be replaced is decided by means of a *(block) replacement strategy* [HP03]. Obviously, a replacement strategy is not needed for a direct mapped cache.

The most commonly used replacement strategies for today's caches are *random* and *least recently used (LRU)*. The random replacement strategy chooses a random cache line from the set of eligible cache lines to be replaced. The LRU strategy replaces the block which has not been accessed for the longest time. According to the principle of locality, it is more likely that a data item which has been accessed recently will be accessed again in the near future. Less common replacement strategies are *least frequently used (LFU)* and *first in, first out (FIFO)*. The former replaces the cache block which has least frequently been used, whereas the latter replaces the data which has been residing in cache for the longest time.

2.4.3 Write Policies

There are two alternatives when data is written to a cache. In the case of a *write through* cache, the corresponding data is written to the cache and to the memory level below; e.g., another level of cache or main memory. In contrast, if the *write back* policy is employed the data is written to cache only. It will be written to the next lower memory level only when it needs to be replaced by another cache block[3].

In addition, there are two alternatives how to handle a *write miss*. Such a miss occurs when a cache line which is not in cache is written to. In the case of a *write allocate* cache, the cache line to be written is first loaded from the lower memory levels and then modified. In contrast, if a *no-write allocate (write around)* cache is used, the cache block is directly modified in the lower level of the

[3]Usually, a *dirty bit* is used to indicate if the cache block has actually been modified and therefore needs to be written to the next lower memory level [HP03].

memory hierarchy. Typically, write back caches use the write allocate policy in the case of write misses, and write through caches use write around. A discussion of advantages and disadvantages of these alternative design approaches can be found in [HP03].

2.4.4 Classification of Cache Misses

Typically, three types of cache misses are distinguished in the literature on computer architecture [HP03].

- *Cold-start miss (compulsory miss):*
  Such a miss occurs if a cache line which has never been accessed before is referenced for the first time.

- *Capacity miss:*
  Such a miss occurs if a data item which was in cache before, but has been evicted from cache due to its limited capacity cannot be loaded from cache.

- *Conflict miss:*
  Conflict misses are those misses which do neither occur due to limited cache capacity nor due to the cache blocks being accessed for the very first time. Instead, they occur if the associativity of the cache is not large enough to resolve mapping conflicts. Hence, conflict misses do not occur in the case of a fully associative cache. The phenomenon of the code performance being heavily spoiled by conflict misses is called *cache thrashing* [GH01].

2.5 Cache Performance Analysis

2.5.1 Code Profiling — Techniques and Tools

Profiling techniques are used in order to analyze the runtime behavior of application codes. In general, the term *profiling* refers to the recording of summary information during the execution of a code. This information may include timing data, statistics on function calls and hardware events, etc. Profiling tools are therefore employed to determine if a code runs efficiently, to expose hot spots and performance bottlenecks, as well as to guide code optimization [GH01]. Ideally, profiling should not introduce any overhead at all; i.e., it should not influence the execution (in particular, the performance behavior) of the code under consideration.

2.5.1.1 Hardware Performance Counters

In order to enable low-overhead performance measurements, most microprocessor manufacturers add dedicated hardware including special-purpose registers to their CPUs to count a variety of events. These registers are called *(hardware) performance counters* [GH01]. The information which can be gathered by the hardware performance counters varies from platform to platform. Typical quantities which can be measured include cache misses and cache hits for various cache levels, pipeline stalls, processor cycles, instruction issues, floating-point operations, and incorrectly predicted branch targets, see also [Wei01].

Generally speaking, data that has been gathered through the use of hardware performance counters must be interpreted very carefully. This is particularly true for data referring to the utilization of the memory hierarchy. For example, a reference to a data item may cause a TLB miss

(due to the current page address not being found in the TLB), although the corresponding data can be quickly retrieved from a virtually addressed cache.

Examples of common profiling tools based on hardware performance counters are the *Performance Counter Library (PCL)* [BM00], the *Performance Application Programming Interface (PAPI)* [BDG+00], *OProfile*[4], as well as the *Digital Continuous Profiling Infrastructure (DCPI)*, which is only available for DEC/Compaq Alpha-based systems running Compaq Tru64 UNIX [ABD+97]. The profiling experiments in this thesis (see Chapter 7) have been carried out using PAPI, OProfile, and DCPI.

The PAPI framework implements three portable interfaces to access the hardware performance counters of a variety of platforms; a fully programmable low-level interface of about 40 functions to access the counters directly, a high-level interface (based on the low-level interface) providing a limited set of elementary library functions to initialize, start, stop, and read the counters for a specified list of hardware events, as well as a graphical tool suite to visualize performance data. The set of available hardware events varies from platform to platform. Note that recent versions of several parallel performance analysis tools such as *VAMPIR (Visualization and Analysis of MPI Resources)* [NAW+96] and *TAU (Tuning and Analysis Utilities)* [MMC96] use the PAPI interface to access hardware performance counters.

OProfile and DCPI employ the technique of *statistical sampling* [WKT04], particularly *event-based sampling*. This means that the corresponding hardware counter is initialized with a certain value; the so-called *sampling period*. Whenever the event to be monitored occurs during the execution of the program, the counter is decremented. As soon as the actual value of the counter reaches 0, an interrupt is raised. The interrupt handler has access to the program counter and can thus assign the interrupt to the current line of source code or even to the current machine instruction. Then, the hardware counter is reset to its initial value and the execution of the original program continues. If the sampling period is chosen large enough, the profiling overhead is only marginal. However, the larger the sampling period, the less accurate results can be expected. Hence, the selection of appropriate sampling periods is generally a nontrivial task and represents a trade-off between overhead and accuracy.

2.5.1.2 Instrumentation

Additional profiling approaches are based on *instrumentation*; i.e., on the direct insertion of measurement code [WKT04]. Tools such as *GNU prof* [FS98] and *ATOM* [ES95], for example, insert calls to monitoring libraries into the program to gather information about certain code regions or to generate trace files. Instrumentation is often used to determine the fraction of CPU time spent in the individual subroutines. However, the library routines may also invoke complex programs themselves (e.g., hardware simulators), modify user-defined event counters, or read hardware performance counters, for example. Various instrumentation techniques have been introduced; e.g., source code instrumentation, compiler-injected instrumentation, instrumentation by binary translation, and runtime instrumentation [WKT04].

Unfortunately, instrumentation often involves a significant amount of overhead and deteriorates the runtime performance of the code. It often increases execution time such that time measurements become useless. Yet, code instrumentation is often used to drive hardware simulations, see Section 2.5.2. In this context, the execution time of the code is not an issue.

[4] See `http://oprofile.sourceforge.net`.

2.5.2 Simulation-Based Cache Performance Analysis

Cache performance data can also be obtained by *cache modeling* and *cache simulation* [GMM97, HKN99], or even by comprehensive *machine simulation* [MRF+00]. See also the presentation and the references to the literature provided in [Wei01]. However, hardware simulation is usually very time-consuming compared to regular program execution. Thus, the underlying cache models and machine models must often be simplified in order to reduce simulation complexity. Consequently, simulation results must always be interpreted carefully and validated thoroughly by comparing them with real performance data. An overview of cache simulation techniques is presented in [WKT04].

The advantage of simulation-based performance analysis and simulation-driven optimizations is that, through the use of suitably simplified machine models, very general and portable code optimizations can be devised. Moreover, with machine simulation, there is no need for accessing special-purpose hardware such as performance counters. Consequently, machine simulators are very flexible and can easily be adapted to model a variety of target platforms. This can be accomplished through the introduction of appropriate hardware parameters into the simulation infrastructure; e.g., cache capacities and cache line lengths.

The Intel IA-32-based runtime instrumentation framework *Valgrind* represents an example of a highly flexible software infrastructure for machine simulation [NS03]. An important advantage of this approach is that it is not necessary to recompile or even to modify the code to be instrumented and analyzed. The Valgrind instrumentation mechanism depends on the use case and can be specified using customized plug-ins. One of the different instrumentation plug-ins for Valgrind is *Cachegrind*, which implements a simulator for two-level cache hierarchies of Intel CPUs, see again [NS03]. We have used Valgrind and Cachegrind in order to simulate the cache behavior of our locality-optimized codes, see Section 7.3.2.4.

Measuring and simulating the cache performance of an application is closely related to the issue of visualizing memory hierarchies on one hand and performance data on the other. A discussion of cache visualization techniques exceeds the scope of this work. We refer to [Wei01] instead. A survey on performance data visualization approaches is again provided in [WKT04].

For an overview of machine-independent locality metrics, which can be employed to measure and to enhance the locality exhibited by an application code in a general fashion, see Section 9.4.

Chapter 3

Numerical Algorithms I — Iterative Linear Solvers

3.1 Preparation: Discretization of PDEs

Many phenomena in science and engineering can be described by mathematical models involving *partial differential equations (PDEs)* together with suitable *initial conditions* and/or *boundary conditions*. While the former specify the behavior of the PDE solution at some initial point in time, the latter prescribe its behavior along the boundary of the computational domain [Bra97].

For example, the formulation of a scalar, elliptic, and linear Dirichlet boundary value problem (BVP) reads as follows:

$$\begin{aligned} Lu &= f \quad \text{in} \quad \Omega \ , \\ u &= g \quad \text{on} \quad \partial\Omega \ , \end{aligned}$$

cf. [Hac96]. Here, L denotes the linear and elliptic differential operator, u denotes the unknown scalar function, and f is the right-hand side. Furthermore, Ω is the domain, $\partial\Omega$ is its boundary, and the function g specifies the values of the solution u along the boundary $\partial\Omega$. Note that, throughout this thesis, we will use the symbols u and f to denote both continuous and discrete quantities. The actual meaning, however, will always become clear from the context.

Probably, the most popular textbook example for an elliptic differential operator L is the negative *Laplace operator* (or *Laplacian*); i.e., $L := -\Delta$, where

$$\Delta := \sum_{i=1}^{d} \frac{\partial^2}{\partial {x_i}^2} \ ,$$

and d denotes the dimension of the problem. This differential operator occurs in the simulation of diffusion processes and temperature distributions, for example [KA00].

The numerical solution of PDEs is a large field of ongoing research. The standard approach is based on

1. an appropriate discretization of the continuous problem including the prescribed boundary conditions (e.g., using finite differences or finite elements) as well as
2. the subsequent numerical solution of the resulting algebraic system of (linear or nonlinear) equations.

We focus on the solution of linear PDEs such that the discretization process yields a system of linear equations. For a detailed introduction to discretization techniques for PDEs, we refer to the literature; e.g., [Bra97, Hac96, KA00].

In most parts of this thesis, we assume that the given PDE is discretized on a *regular (structured) grid*[1]. Such a grid is characterized by the property that each interior node has a fixed number of neighbors; i.e., a fixed *connectivity*. If the partial derivatives occurring in the differential operator L are approximated by finite differences, each grid node representing an unknown solution value can be assigned a *stencil* of coefficients. The coefficients of a stencil specify how the corresponding unknown is influenced by the values at its neighbors in the grid. The size of the stencil depends on the approximation orders of the underlying finite difference formulae. Stencils with similar structures occur in the case of finite element approaches as long as the underlying basis functions have local support.

Each of these volume discretization approaches leads to a *sparse* system of linear equations. This means that the system matrix is characterized by a high percentage of zero entries, see also Section 3.2.1.4. The number of unknowns in a realistic simulation may easily exceed 10^6, even by several orders of magnitude. Today's large-scale parallel computations involve the solution of linear systems with up to 10^{10} unknowns. Therefore, the resulting linear system is typically solved numerically, see Section 3.2. Hereby, the structure of the system matrix depends on the numbering of the unknowns; i.e., on the ordering of the equations in the linear system. We refer to Section 3.2.3 for examples.

At this point, it is important to note that all data locality optimizations we will present in Chapters 5 and 6 exploit the structural locality exhibited by the compact discretization stencils that we focus on. With respect to our cache performance optimizations, it is irrelevant if these stencils result from the application of finite difference approximations or finite element techniques. Certainly, the underlying discretization approach influences numerical properties, in particular the accuracy of the discrete numerical solution.

The following section provides a brief overview of elementary numerical algorithms for the solution of systems of linear equations. *Multigrid methods*, which form a class of more sophisticated numerical schemes, will be addressed separately in Section 3.3.

3.2 Elementary Linear Solvers

3.2.1 Overview: Direct Methods vs. Iterative Methods

3.2.1.1 Preparations

The algorithms we discuss in this section can be used to compute the solution of a system

$$Au = f \tag{3.1}$$

of linear equations. In the following,

$$A := (a_{i,j})_{1 \le i,j \le n} \in \mathbf{R}^{n \times n}$$

is a regular matrix,

$$f := (f_1, f_2, \ldots, f_n)^T \in \mathbf{R}^n$$

[1] Cache performance optimizations for computations on *unstructured grids* will be addressed in Section 9.1.3.

is the right-hand side, and

$$u := (u_1, u_2, \ldots, u_n)^T \in \mathbf{R}^n$$

represents the vector of unknowns. Since A is regular, the exact solution $u^* := A^{-1}f$ exists and is unique. It is denoted as

$$u^* := (u_1^*, u_2^*, \ldots, u_n^*)^T \in \mathbf{R}^n \ .$$

One can distinguish two basic classes of numerical algorithms for the solution of (3.1); *direct* methods and *iterative* methods. In the following, we will give a brief characterization of these two classes of algorithms and refer the reader to a variety of standard textbooks on numerical analysis; e.g., [GL98, HS02, Sto99, SB00, Sch97, Ueb97b].

3.2.1.2 Direct Methods

Most direct methods are based on Gaussian elimination and compute factorizations of A or of some matrix $\tilde{A}$ which is obtained by applying elementary permutations to A in the context of pivoting. See [GL98], for example. Direct methods have in common that the numerical solution of (3.1) is computed using a predictable finite number of arithmetic operations, which depends on the dimension n of the linear system (3.1) and the structure of the matrix A. Typical direct methods employ LU factorizations, QR factorizations, or Cholesky decompositions in the symmetric positive definite case. For detailed discussions concerning accuracy and stability of direct methods, we particularly refer to [Hig02].

A large selection of classical direct methods for the solution of systems of linear equations is part of the freely available numerical software library *LAPACK*. See Section 9.2 and [Ueb97b] for an overview as well as the user guide [ABB+99] for details on LAPACK. A description of cache-conscious block algorithms for the direct solution of systems of linear equations in LAPACK can be found in Section 9.2.3.

For special cases, very efficient direct solution methods based on the recursive reduction of the system matrix A have been proposed. These methods have in common that they exploit the regular structure of A; e.g., Buneman's algorithm [SB00]. Other direct methods which can efficiently be applied for special problems employ a Fourier decomposition of the right-hand side f in order to assemble the Fourier decomposition of the exact solution u^* afterwards. These algorithms are typically based on the fast Fourier transform (FFT) [HS02, Sch97].

3.2.1.3 Iterative Methods

In this thesis, we will focus on iterative methods for the numerical solution of (3.1), see Section 3.2.1.4. Standard textbooks on numerical analysis usually contain introductions to iterative methods for linear systems; e.g., [HS02, Sch97, SB00]. Extensive studies of this class of numerical algorithms can be found in [GL98, Gre97, Hac93, Var62, You71].

Based on an initial guess $u^{(0)}$, an iterative algorithm computes a series

$$u^{(0)} \to u^{(1)} \to u^{(2)} \to \ldots$$

of numerical approximations

$$u^{(k)} := \left(u_1^{(k)}, u_2^{(k)}, \ldots, u_n^{(k)}\right)^T \in \mathbf{R}^n$$

which converges to the exact solution u^* of the linear system (3.1).

We follow the notation in [Hac93] and define an *iterative method* to be a mapping

$$\Phi : \mathbf{R}^n \times \mathbf{R}^n \to \mathbf{R}^n \ .$$

In particular, we write

$$u^{(k+1)} = \Phi\left(u^{(k)}, f\right) \ ,$$

in order to express that any new approximation $u^{(k+1)}$, $k \geq 0$, is computed from the previous approximation $u^{(k)}$ and from the right-hand side f of (3.1). Of course, any iterative process will depend on the matrix A as well, which is not expressed explicitly in this formalism.

The iterative process terminates as soon as a given *stopping criterion* has been reached and the current numerical approximation is therefore considered accurate enough. Typical approaches are based on the *residual vector* (or simply the *residual* or the *defect*) $r^{(k)} \in \mathbf{R}^n$, which is defined as

$$r^{(k)} := f - Au^{(k)} \ , \tag{3.2}$$

and cause the iterative process to terminate as soon as some vector norm $||.||$ of the residual $r^{(k)}$ has reached a prescribed tolerance. Other approaches cause the iterative process to terminate as soon as some vector norm $||\cdot||$ of the difference $u^{(k+1)} - u^{(k)}$ of two successive approximations $u^{(k)}$ and $u^{(k+1)}$ has fallen below some prescribed tolerance.

The choice of an appropriate stopping criterion is usually a nontrivial task. The aforementioned stopping criteria both have disadvantages. If, for example, the linear system (3.1) is ill-conditioned, a small residual $r^{(k)}$ does not necessarily imply that the actual *(algebraic) error*

$$e^{(k)} := u^* - u^{(k)} \tag{3.3}$$

is small as well [Sch97], see also Section 3.3.4. Additionally, if the convergence is slow, any two successive approximations $u^{(k)}$ and $u^{(k+1)}$ may be very close to each other, although each of them might differ enormously from the exact solution u^*.

All iterative schemes we will discuss in this work are assumed to be *consistent*; i.e., the exact solution u^* of (3.1) is a fixed point of the mapping Φ:

$$u^* = \Phi\left(u^*, f\right) \ ,$$

see [Hac93]. Moreover, all iterative methods we will focus on (except for the advanced adaptive multigrid schemes we will present in Chapter 8) are *linear*, which means that there are matrices $M, N \in \mathbf{R}^{n\times n}$, such that we can write

$$u^{(k+1)} = \Phi\left(u^{(k)}, f\right) = Mu^{(k)} + Nf \ . \tag{3.4}$$

In addition, some authors introduce the term *stationary* in order to indicate that neither M nor N depend on the current iterate $u^{(k)}$ or on the current iteration index k [Sch97]. Typically, M is called the *iteration matrix* of the linear iterative method defined by the mapping Φ, while N is a so-called *approximate inverse* to A, $N \approx A^{-1}$. It can be shown that, in the case of a consistent iterative scheme, we have

$$M = I - NA \ , \tag{3.5}$$

where, as usual, $I \in \mathbf{R}^{n\times n}$ denotes the identity [Hac93]. Note that, if $N = A^{-1}$ holds, we find that $M = 0$, which in turn means that the method converges to the exact solution u^* within a single iteration.

Using (3.5), we can derive an equivalent representation of (3.4):

$$u^{(k+1)} = Mu^{(k)} + Nf = u^{(k)} + Nr^{(k)} \ , \tag{3.6}$$

where $r^{(k)}$ stands for the residual corresponding to the k-th iterate $u^{(k)}$, cf. Definition (3.2). Since, if $u^{(k)} = u^*$ holds, the residual $r^{(k)}$ vanishes, (3.6) confirms that u^* is actually a fixed point of the iteration; i.e.,

$$u^* = Mu^* + Nf \ . \tag{3.7}$$

From the definition of the algebraic error in (3.3) and from (3.7), it follows that

$$e^{(k+1)} = Me^{(k)} \ , \tag{3.8}$$

which in turn implies that

$$e^{(k)} = M^k e^{(0)} \ . \tag{3.9}$$

These two relations represent an important property of linear iterative methods. In particular, they allow their convergence analysis to be derived from the spectral properties of the corresponding iteration matrix M. Again, we particularly refer to [Hac93] for a comprehensive analysis.

Whether a linear iterative method converges to the exact solution u^* of the linear system (3.1) for every initial guess $u^{(0)}$ depends on the *spectral radius* $\rho(M)$ of the iteration matrix M, which is defined as follows:

$$\rho(M) := \max\{|\lambda_i|; \lambda_i \in \sigma(M)\} \ . \tag{3.10}$$

As usual, $\sigma(M)$ denotes the *spectrum* of M; i.e., the set of all eigenvalues of M. An elementary result from numerical linear algebra states that any linear iterative method with iteration matrix M converges to the exact solution u^* of the linear system (3.1) for every initial guess $u^{(0)}$ if and only if $\rho(M) < 1$. This result follows from (3.9) [Hac93]. In particular, if $\rho(M) < 1$, but still $\rho(M) \approx 1$, the iteration will generally converge very slowly, because the error components corresponding to the eigenvalue λ of M with $|\lambda| = \rho(M)$ are only reduced very inefficiently.

In order that we can discuss the linear iterative methods in Sections 3.2.2, 3.2.3, and 3.2.4 using matrix-vector notation, we need to introduce some further notation. We split the matrix A into three matrices as follows:

$$A = D - L - U \ . \tag{3.11}$$

Here, $D = \operatorname{diag}(a_{1,1}, a_{2,2}, \ldots, a_{n,n})$ is a diagonal matrix that contains the diagonal entries of A. Since A is regular, we may assume without loss of generality that $a_{i,i} \neq 0$, $1 \leq i \leq n$, and that D is therefore invertible [Sch97]. Furthermore, $-L$ is a strictly lower triangular matrix which contains the strictly lower triangular part of A, whereas $-U$ is a strictly upper triangular matrix which contains the strictly upper triangular part of A. We will refer to this notation shortly.

There is another class of algorithms that must be mentioned in the context of the numerical solution of linear systems; the *Krylov subspace methods* such as the method of conjugate gradients, BiCGStab, and GMRES, for example. From a theoretical point of view, these algorithms are direct methods as well because the number of steps which need to be executed in order to solve the linear system (3.1) numerically is bounded by its order n. However, there are two reasons why these algorithms are typically considered as (nonlinear) iterative schemes. Firstly, n may be a relatively large number (sometimes up to more than 10^9), such that it is impractical to perform n update steps on the initial guess $u^{(0)}$. Secondly, due to finite precision arithmetic of computer hardware and the resulting numerical round-off, this upper bound for the number of steps becomes again a pure theoretical property. Krylov subspace methods are commonly used in combination with

appropriate *preconditioning* techniques, which target the optimization of the spectral properties of the system matrix and thus the convergence behavior of the respective method.

A comprehensive discussion of Krylov subspace algorithms is beyond the scope of this thesis. Instead, we refer to [Gre97, Hac93] for an introduction to the theory. A discussion of a cache-optimized variant of the GMRES algorithm based on sparse matrix-multivector multiplies can be found in [BDJ03].

3.2.1.4 Why We Focus on Iterative Methods

The linear systems we deal with result from the discretization of elliptic PDEs using finite differences or finite elements, see Section 3.1. The corresponding matrices are typically large as well as sparse; i.e., the number of nonzeros is only of order $\mathcal{O}(n)$.

In the course of the Gaussian elimination process, direct methods introduce *fill-in*. This means that more and more zeros in the matrix, which do not have to be stored if suitable storage schemes for sparse matrices are used [BBC+94], need to be replaced by nonzeros, which must actually be stored and, moreover, eliminated again later on. For reasonable problem sizes, this results in enormous memory consumption and, even worse, often unacceptable computational complexity. Even if special structural properties of the matrix (such as limited bandwidth, for example) are exploited, this argument still holds.

Recent research has focused on the development of direct solvers involving large sparse matrices. Like most direct solvers, the algorithms belonging to this class of *sparse direct solvers* split the system matrix into triangular factors and then employ a forward-backward substitution step to compute the solution. However, sparse matrix ordering techniques are additionally integrated. They aim at minimizing both the storage by reducing the fill-in as well as the resulting computational work [DKP99].

However, due to the enormous complexity of direct solvers (except for the efficient direct solvers for special problems mentioned in Section 3.2.1.2), large sparse systems of linear equations are often solved using iterative numerical algorithms. These methods are characterized by two major advantages. Firstly, their memory requirements are comparatively low (typically of the order $\mathcal{O}(n)$) and remain constant in the course of the solution process. Secondly, their computational complexities are often much lower than those of direct solution methods.

As a prominent example, we consider the numerical solution of the standard 3D model problem $-\Delta u = f$ on the unit cube, involving Dirichlet boundary conditions. We assume that the Laplacian is discretized on an equidistant grid using second-order finite differences. This approach leads to a symmetric positive definite system matrix A based on constant 7-point stencils. If we assume a lexicographic ordering of the n unknowns, the bandwidth of A will be approximately $n^{\frac{2}{3}}$. Since, in general, the computation of the Cholesky decomposition of a banded matrix with bandwidth b requires $\mathcal{O}(nb^2)$ operations, the Cholesky decomposition of A requires $\mathcal{O}(n^{\frac{7}{3}})$ operations, and $\mathcal{O}(n^{\frac{5}{3}})$ matrix entries must be stored. The subsequent forward-backward substitution step generally needs $\mathcal{O}(nb)$ operations, which reduces to $\mathcal{O}(n^{\frac{5}{3}})$ in our case.

In contrast, in the case of this model problem, the execution of a single SOR operation requires $\mathcal{O}(n)$ operations. Furthermore, $\mathcal{O}(n)$ SOR iterations are needed to solve the linear system, cf. Section 9.4.3.2. Therefore, the overall computational complexity of the SOR method is of order $\mathcal{O}(n^2)$. From this, we can see that the decomposition step of Cholesky's method alone is asymptotically more expensive than the entire iterative solution process based on the SOR algorithm. Moreover, as we have mentioned above, the memory consumption is only of the order $\mathcal{O}(n)$ as well.

Still, there is a downside to any iterative method; its convergence behavior depends on the properties of the system matrix A. Hence, iterative schemes are not as robust and universally applicable as direct methods. In general, it is a nontrivial task to determine if an iterative algorithm converges at all, to determine the iterative method which converges fastest for a given problem, and to derive quantitative estimates concerning the convergence behavior. Much ongoing research focuses on the extension of the theoretical foundations and, as a consequence, on the applicability of iterative schemes to the solution of large systems of linear equations [GL98, Gre97, Hac93, Var62, You71].

In the following, we will briefly introduce a set of basic iterative schemes as well as the fundamentals of multigrid methods. We will repeatedly refer to the subsequent descriptions in the remainder of this thesis.

3.2.2 Jacobi's Method

Jacobi's method is given by the following update scheme:

$$u_i^{(k+1)} = a_{i,i}^{-1} \left(f_i - \sum_{j \neq i} a_{i,j} u_j^{(k)} \right) \quad , \tag{3.12}$$

where $1 \leq i \leq n$. Note that, in order to compute any component $u_i^{(k+1)}$ of the new approximation $u^{(k+1)}$, only components of the previous approximation $u^{(k)}$ are accessed. This means that, in an implementation of Jacobi's method, $u_i^{(k)}$ may only be overwritten by $u_i^{(k+1)}$, if $u_i^{(k)}$ is not required for computing further components of $u^{(k+1)}$. In other words, temporary storage space must be provided in order to preserve all data dependences which are defined by the algorithm. This property of Jacobi's method further implies that the numbering of the unknowns is irrelevant and, therefore, does not influence the numerical results. In particular, this property also simplifies the parallelization of Jacobi's method.

It is easy to see that, using the notation from (3.11), the update scheme for Jacobi's method given by (3.12) can be written in matrix-vector form as follows:

$$u^{(k+1)} = D^{-1}(L+U)u^{(k)} + D^{-1}f \quad ,$$

Using the notation introduced in (3.4), we therefore find that

$$M_{\text{Jac}} = D^{-1}(L+U)$$

and

$$N_{\text{Jac}} = D^{-1} \quad ,$$

which shows that Jacobi's method is linear.

3.2.3 The Method of Gauss-Seidel

The *method of Gauss-Seidel* is given by the following update scheme:

$$u_i^{(k+1)} = a_{i,i}^{-1} \left(f_i - \sum_{j<i} a_{i,j} u_j^{(k+1)} - \sum_{j>i} a_{i,j} u_j^{(k)} \right) \quad , \tag{3.13}$$

where $1 \leq i \leq n$. Note that, in order to compute any component $u_i^{(k+1)}$ of the new iterate $u^{(k+1)}$, we are required to use new components $u_j^{(k+1)}$ for all unknowns with indices $j < i$; i.e., for all

unknowns whose approximations have already been updated in the course of the current iteration $k+1$. For those unknowns with $j > i$, however, we have to use the components $u_j^{(k)}$ from the previous iteration k, since new approximations from iteration $k+1$ are not available yet.

Consequently, in an implementation of the method of Gauss-Seidel, we may overwrite components as soon as they are available such that, for all further update steps in the course of the current iteration, the latest values of all components of the solution vector will be accessed. This means that no temporary storage space is needed in order to maintain data dependences.

Furthermore, in contrast to Jacobi's method, the ordering of the unknowns plays an important role in the context of the method of Gauss-Seidel. Different orderings generally imply different numerical results occurring in the course of the computation. However, there is some theory ensuring that, under certain assumptions, the asymptotic convergence rate of the iterative algorithm (i.e., the spectral radius $\rho(M)$ of the iteration matrix M) does not depend on the ordering of the unknowns [Hac93]. In a multigrid setting, however, the ordering of the unknowns can significantly influence the numerical efficiency of Gauss-Seidel-type smoothers, see Section 3.3.4.

Additionally, the ordering of the unknowns defines the data dependences that need to be preserved. In particular, these data dependences strongly influence how efficiently the code can be parallelized, and how it can be optimized by the computer hardware (e.g., out-of-order execution), the compiler (e.g., software pipelining, loop interchange), and, of course, the programmer.

Typical orderings of the unknowns are the *lexicographic ordering* and the *red-black ordering* [BHM00], which many of our data access optimization techniques will be applied to, see Chapter 6. In a lexicographic ordering, the unknowns are numbered one after the other, starting in one corner of the grid and traversing the latter until the opposite corner is reached. In a red-black ordering, each unknown whose grid indices (corresponding to the coordinates of the respective grid node) yield an even sum is considered red. Otherwise, it is considered black. This definition is valid for 1D (in which case each grid node is characterized by a single index), 2D, and 3D grids. Figure 3.1 illustrates a regular 2D grid with a red-black ordering of the unknowns. The outermost layer of grid nodes (shaded gray) contains Dirichlet boundary points.

In the context of a red-black Gauss-Seidel method, all red nodes are updated first. Afterwards, all black nodes are updated. The basic pattern of this method is demonstrated in Algorithm 3.1.

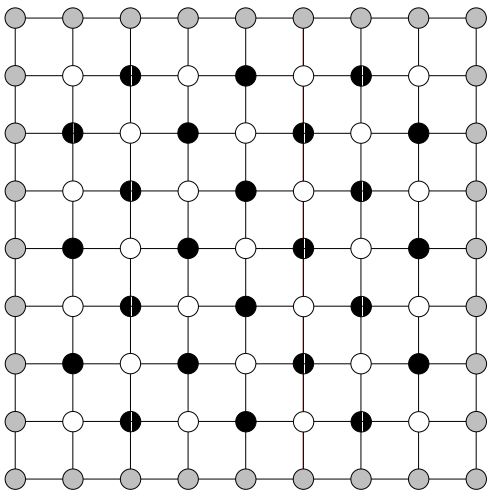


Figure 3.1: Red-black ordering of the unknowns in a regular 2D grid.

Algorithm 3.1 Red-black Gauss-Seidel — basic structure.

```
1: for i = 1 to i_max do
2:    for all red nodes i_r do
3:       update(i_r)
4:    end for
5:    for all black nodes i_b do
6:       update(i_b)
7:    end for
8: end for
```

For the sake of simplicity, we have introduced the variable $i_{\max}$ to denote the total number of iterations to be performed. If the discretization of the underlying PDE is based on 5-point stencils in 2D or, correspondingly, on 7-point stencils in 3D, the numerical solution at any red node only depends on the values at its black neighbors, and vice versa. This means that, in each Gauss-Seidel iteration, the red unknowns can be updated in any order and, subsequently, the black unknowns can be updated in any order as well. Hence, the red-black Gauss-Seidel method exhibits a high potential for parallelism and for cache performance optimizations. Such cache performance optimizations for red-black Gauss-Seidel will be presented in Chapter 6.

Analogous to the case of Jacobi's method, it is obvious that, using matrix-vector notation and the splitting of A given by (3.11), the Gauss-Seidel update scheme can be written as follows:

$$u^{(k+1)} = (D - L)^{-1} U u^{(k)} + (D - L)^{-1} f \ ,$$

Likewise, recalling the notation from (3.4), we obtain

$$M_{\mathrm{GS}} = (D - L)^{-1} U$$

and

$$N_{\mathrm{GS}} = (D - L)^{-1} \ ,$$

which shows that the method of Gauss-Seidel is linear as well. Note that $(D - L)^{-1}$ exists since $a_{i,i} \neq 0$, $1 \leq i \leq n$. Hence, $D - L$ has only nonzeros on its diagonal, which in turn implies that $\det(D - L) \neq 0$.

The data dependences of a single Gauss-Seidel iteration are also reflected by the fact that the corresponding iteration matrix M_{GS} involves the inverse of the lower triangular matrix $D-L$, which is again a lower triangular matrix. In contrast, Jacobi's method involves only the inverse of the diagonal matrix D. This indicates again that, in the case of Jacobi's method, no new component $u_i^{(k+1)}$ depends on any other new component $u_j^{(k+1)}$, but only on components $u_j^{(k)}$ from the previous iteration k.

3.2.4 Jacobi Over-Relaxation and Successive Over-Relaxation

In many cases, the convergence speed of Jacobi's method and the method of Gauss-Seidel can be enhanced by introducing a relaxation parameter ω. This parameter ω introduces a weighting of the corresponding previous and new iterates.

In a single-grid setting, the idea behind this approach is to reduce the spectral radius of the iteration matrix; i.e., $\rho(M_{\mathrm{Jac}})$ in the case of Jacobi's method and $\rho(M_{\mathrm{GS}})$ in the case of Gauss-Seidel. In a multigrid method, algorithms such as Jacobi's method and the method of Gauss-Seidel are often used as *smoothers*, see Section 3.3. In these cases, relaxation parameters are

typically introduced to optimize the smoothing properties of the standard algorithms [BHM00, TOS01, Yav96].

The method of *Jacobi over-relation (JOR, weighted Jacobi's method, damped Jacobi's method)* is given by the following update scheme:

$$u_i^{(k+1)} = (1-\omega)u_i^{(k)} + \omega a_{i,i}^{-1}\left(f_i - \sum_{j\neq i} a_{i,j}u_j^{(k)}\right) \ , \tag{3.14}$$

where $1 \leq i \leq n$. All remarks concerning the implementation of Jacobi's method carry over to the JOR method.

The notation we have introduced in (3.11) also allows the JOR method to be written in matrix-vector form:

$$u^{(k+1)} = \left[(1-\omega)I + \omega D^{-1}(L+U)\right] u^{(k)} + \omega D^{-1} f \ ,$$

Using the notation introduced in (3.4), we therefore obtain

$$M_{\mathrm{JOR}} = (1-\omega)I + \omega D^{-1}(L+U)$$

and

$$N_{\mathrm{JOR}} = \omega D^{-1} \ ,$$

which shows that the JOR method is also linear.

The method of *successive over-relation (SOR)* is given by the following update scheme:

$$u_i^{(k+1)} = (1-\omega)u_i^{(k)} + \omega a_{i,i}^{-1}\left(f_i - \sum_{j<i} a_{i,j}u_j^{(k+1)} - \sum_{j>i} a_{i,j}u_j^{(k)}\right) \ , \tag{3.15}$$

where $1 \leq i \leq n$. Correspondingly, all remarks concerning the implementation of the method of Gauss-Seidel carry over to the SOR method. In particular, the ordering of the unknowns is essential.

Using the notation from (3.11) once more, we obtain

$$u^{(k+1)} = (D-\omega L)^{-1}\left[(1-\omega)D + \omega U\right] u^{(k)} + \omega(D-\omega L)^{-1} f \ ,$$

Recalling the notation from (3.4), we get

$$M_{\mathrm{SOR}} = (D-\omega L)^{-1}\left[(1-\omega)D + \omega U\right]$$

and

$$N_{\mathrm{SOR}} = \omega(D-\omega L)^{-1} \ .$$

Hence, the SOR method is a linear iteration scheme as well.

A presentation of the standard theory concerning the choice of ω for the JOR method and the SOR method is beyond the scope of this thesis. Many results can be found in the literature on iterative methods, see again [GL98, Gre97, Hac93, Var62, You71].

3.2.5 Remarks

It is a nontrivial task to determine the best iterative method to solve a given problem. It is even nontrivial to define what it means for an iterative method to be the *best* among a variety of alternatives. Such a measure of quality must necessarily cover both theoretical convergence characteristics as well as practical implementation aspects and their impact on the runtime performance of the codes.

In the context of linear iterative schemes, the discussion of theoretical convergence issues is primarily concerned with the investigation of the spectral properties of the iteration matrices, but also with the question of how to choose reasonable stopping criteria, cf. Section 3.2.1.3. If, for example, the method of Gauss-Seidel and a multigrid method are used to solve a linear system until some norm of the residual has fallen below a given threshold, the multigrid method will typically yield a more accurate result since it reduces the error uniformly over its entire frequency spectrum, while the method of Gauss-Seidel is not able to eliminate the low-frequency error components efficiently, see Section 3.3.

Even the determination of the best among the elementary iterative methods introduced in Sections 3.2.2, 3.2.3, and 3.2.4 is often not trivial and depends on the properties of the matrix A in (3.1).

Comprehensive surveys of theoretical convergence results concerning iterative algorithms for linear systems are provided in [GL98, Gre97, Hac93, Var62, You71]. A selection of introductory results can also be found in [Sch97]. Problems in computer science and computer engineering such as reliability analysis and performance evaluation are often modeled using stochastic processes; e.g., Markov chains. Their analysis also involves the numerical solution of large sparse linear systems using iterative algorithms. A survey of theoretical results for this special class of singular matrices as well as a variety of examples for such applications are presented in [Ste94].

3.3 Multigrid Methods

3.3.1 Overview

It has been shown that multigrid methods are among the fastest numerical algorithms for solving large sparse systems of linear equations that arise from appropriate discretizations of elliptic PDEs. Much research has focused and will continue to focus on the design of multigrid algorithms for a large variety of application areas. Hence, there is a large and constantly growing body of literature.

We generally refer to the earlier publications such as [Bra77, Bra84, Hac85], for example, as well as to the recent and comprehensive overview provided in [TOS01]. The latter contains a long list of further references. An easy-to-read introduction to multigrid methods can be found in [BHM00]. This textbook contains many illustrations in order to picture and to clarify the mathematical theory. Additionally, a detailed model problem analysis as well as an introduction to Fourier techniques can be found in the early publication [ST82]. A comprehensive list of multigrid references, which is constantly being updated, can be found in the BibTEX file `mgnet.bib` [DD].

Multigrid methods form a rich class of iterative algorithms for large systems of linear equations, which is characterized by optimal complexity. For a large class of elliptic PDEs, multigrid algorithms can be devised which require $\mathcal{O}(n)$ floating-point operations in order to solve the corresponding linear system with n degrees of freedom up to discretization accuracy. In this part of our thesis, we give a brief overview of the multigrid principle. We provide just as many details as necessary in order to understand the multigrid implementations whose optimizations we will discuss in Chapters 5 to 7. Further aspects of multigrid methods will be presented in Chapter 8 in the context of inherently cache-aware multilevel algorithms and patch adaptivity.

Generally speaking, all *multilevel algorithms* follow the same fundamental design principle. A given problem is solved by integrating different levels of resolution into the solution process. During this process, the contributions of the individual levels are combined appropriately in order to form the required solution. This is particularly true for *multigrid algorithms* which can be seen

as multilevel approaches towards the solution of grid-based problems in numerical mathematics. These problems cover the solution of linear algebraic systems and eigenvalue problems arising from the discretization of PDEs [Sch91]. The discussion of multigrid methods for eigenvalue problems, however, is beyond the scope of this thesis.

In the classical sense, multigrid methods involve a hierarchy of computational grids of different resolutions and can therefore be considered geometrically motivated. This approach has led to the notion of *geometric multigrid (GMG)*. In contrast, later research has additionally addressed the development and the analysis of *algebraic multigrid (AMG)* methods, which target a multigrid-like iterative solution of linear systems without using geometric information from a grid hierarchy, but only the original linear system itself[2]. For an introduction to AMG, see [BHM00, Stü01] and the references given therein.

3.3.2 Preparations

In this section, we will motivate the principles of a basic multigrid scheme. For simplicity, we consider the example of a scalar boundary value problem

$$Lu = f \quad \text{in} \quad \Omega \ , \tag{3.16}$$

defined on the interval of unit length (i.e., $\Omega := (0,1)$), on the unit square (i.e., $\Omega := (0,1)^2$), or on the unit cube (i.e., $\Omega := (0,1)^3$), see also Section 3.1. L denotes a second-order linear elliptic differential operator. We assume Dirichlet boundary conditions only; i.e.,

$$u = g \quad \text{on} \quad \partial\Omega \ . \tag{3.17}$$

We concentrate on the case of an equidistant regular grid. As usual, we use h to denote the mesh width in each dimension. Hence, $n_{\text{dim}} := h^{-1}$ represents the number of subintervals per dimension. In the 1D case, the grid nodes are located at positions

$$\{x = ih;\ 0 \le i \le n_{\text{dim}}\} \subset [0,1] \ .$$

In the 2D case, they are located at positions

$$\{(x_1, x_2) = (i_1 h, i_2 h);\ 0 \le i_1, i_2 \le n_{\text{dim}}\} \subset [0,1]^2 \ ,$$

and in the 3D case, the node positions are given by

$$\{(x_1, x_2, x_3) = (i_1 h, i_2 h, i_3 h);\ 0 \le i_1, i_2, i_3 \le n_{\text{dim}}\} \subset [0,1]^3 \ .$$

Consequently, the grid contains $n_{\text{dim}} + 1$ *nodes* per dimension. Since the outermost grid points represent Dirichlet boundary nodes, the corresponding solution values are fixed. Hence, our grid actually comprises $n_{\text{dim}} - 1$ *unknowns* per dimension.

Our presentation is general enough to cover the cases of finite differences as well as finite element discretizations involving equally sized line elements in 1D, square elements in 2D, and cubic elements in 3D, respectively. We focus on the case of *standard coarsening* only. This means that the mesh width H of any coarse grid is obtained as $H = 2h$, where h denotes the mesh width of the next finer grid. See [TOS01] for an overview of alternative coarsening strategies such as red-black coarsening and semi-coarsening, for example.

The development of multigrid algorithms is motivated by two fundamental and independent observations which we will describe in the following; the equivalence of the original equation and the residual equation as well as the convergence properties of basic iterative solvers.

[2]The term *algebraic multigrid* may thus appear misleading, since an ideal AMG approach would dispense with any computational grid.

3.3.3 The Residual Equation

The discretization of the continuous problem given by (3.16), (3.17) yields the linear system

$$A_h u_h = f_h \ , \tag{3.18}$$

where A_h represents a sparse nonsingular matrix, see Section 3.1. Again, the exact solution of this linear system is explicitly denoted as u_h^*, while u_h stands for an approximation to u_h^*. If necessary, we add superscripts to specify the iteration index; e.g., $u_h^{(k)}$ is used to explicitly denote the k-th iterate, $k \geq 0$. In the following, we further need to distinguish between approximations on grid levels with different mesh widths. Therefore, we use the indices h and H to indicate that the corresponding quantities belong to the grids of sizes h and H, respectively.

As usual, the *residual* r_h corresponding to the approximation u_h is defined as

$$r_h := f_h - A_h u_h \ . \tag{3.19}$$

Recall that the *(algebraic) error* e_h corresponding to the current approximation u_h is given by

$$e_h := u_h^* - u_h \ , \tag{3.20}$$

cf. Definitions (3.2) and (3.3) in Section 3.2.1.3. From these definitions, we obtain

$$A_h e_h = A_h \left(u_h^* - u_h\right) = A_h u_h^* - A_h u_h = f_h - A_h u_h = r_h \ ,$$

which relates the current error e_h to the current residual r_h. Hence, the *residual (defect) equation* reads as

$$A_h e_h = r_h \ . \tag{3.21}$$

Note that (3.21) is equivalent to (3.18), and the numerical solution of both linear systems is equally expensive since they involve the same system matrix A_h. The actual motivation for these algebraic transformations is not yet obvious and will be provided next.

3.3.4 Convergence Behavior of Elementary Iterative Methods

There is a downside to all elementary iterative solvers that we have introduced in Section 3.2. Generally speaking, when applied to large sparse linear systems arising in the context of numerical PDE solution, they cannot efficiently reduce *slowly oscillating (low-frequency, smooth)* discrete Fourier components of the algebraic error. However, they often succeed in efficiently eliminating *highly oscillating (high-frequency, rough)* error components [BHM00].

This behavior can be investigated analytically in detail as long as certain model problems (e.g., involving standard discretizations of the Laplace operator Δ) as well as the standard iterative schemes presented in Section 3.2 are used. This analysis is based on a decomposition of the initial error $e_h^{(0)}$. This vector is written as a linear combination of the eigenvectors of the corresponding iteration matrix M. In the case of the model problems under consideration, these eigenvectors correspond to the discrete Fourier modes. Using either (3.8), which describes how the error evolves from iteration to iteration, or (3.9), which expresses the remaining algebraic error after the first k iterations, it is possible to examine how fast the individual Fourier modes are eliminated.

As long as standard model problems are considered, it can be shown that the spectral radius $\rho(M)$ of the iteration matrix M behaves like $1 - \mathcal{O}(h^2)$ for the method of Gauss-Seidel, Jacobi's method, and JOR, and like $1 - \mathcal{O}(h)$ for SOR with optimal relaxation parameter, respectively

[SB00]. This observation indicates that, due to their slow convergence rates, these methods are hardly applicable to large realistic problems involving small mesh widths h. Recall the remarks from Section 3.2.1.3 concerning the spectral radius $\rho(M)$ of the iteration matrix.

A closer look reveals that the smooth error modes, which cannot be eliminated efficiently, correspond to those eigenvectors of M which belong to the relatively large eigenvalues; i.e, to the eigenvalues close to 1[3]. This fact explains the slow reduction of low-frequency error modes. In contrast, the highly oscillating error components often correspond to those eigenvectors of M which belong to relatively small eigenvalues; i.e., to the eigenvalues close to 0. As we have mentioned previously, these high-frequency error components can thus often be reduced quickly and, after a few iterations only, the smooth components dominate the remaining error.

Note that whether an iterative scheme has this so-called *smoothing property* depends on the problem to be solved. For example, Jacobi's method cannot be used in order to eliminate high-frequency error modes quickly, if the discrete problem is based on a standard finite difference discretization of the Laplacian, see [BHM00]. In this case, high-frequency error modes can only be eliminated efficiently, if a suitable relaxation parameter is introduced; i.e., if the JOR scheme is employed instead, cf. Section 3.2.4.

This fundamental property of basic iterative schemes is related to the *condition number* $\kappa(A_h)$ of the matrix A_h, which is given by

$$\kappa(A_h) := ||A_h||\ ||A_h^{-1}|| \ .$$

Note that $\kappa(A_h)$ depends on the underlying matrix norm $||\cdot||$, see [GL98]. Generally speaking, if a matrix is ill-conditioned (i.e., if it has a large condition number), a small residual does not necessarily imply that the algebraic error is small as well [Sch97]. Consequently, the residual-based representation (3.6) of any linear iterative scheme illustrates the slow convergence (i.e., the slow reduction of the algebraic error) that can commonly be observed for ill-conditioned problems.

For a typical discretization of any second-order elliptic operator, the condition number $\kappa(A_h)$ of the resulting system matrix A_h — in any matrix norm $||\cdot||$ — can be shown to be of order $\mathcal{O}(h^{-2})$, see [Gre97]. This means that the condition becomes worse as the mesh size h decreases. Norm estimates which involve the condition number $\kappa(A_h)$ of A_h can therefore be used as well in order to examine and to illustrate the convergence behavior of elementary iterative methods.

In addition, the numerical efficiency of Gauss-Seidel-type smoothers depends on the update order of the unknowns, cf. Section 3.2.3. Generally, it is a nontrivial task to determine the best update order. For Poisson problems which are discretized using standard finite difference techniques, for example, it has been shown that a red-black Gauss-Seidel smoother performs better than a Gauss-Seidel scheme based on a lexicographic ordering of the unknowns [ST82]. In the case of anisotropic operators, line smoothing (in 2D) and plane smoothing (in 3D) are often used. If the anisotropy is only moderate, appropriate relaxation parameters can improve the numerical efficiency of a standard red-black SOR smoother [Yav96]. See [TOS01] for further details and examples.

3.3.5 Aspects of Multigrid Methods

3.3.5.1 Coarse-Grid Representation of the Residual Equation

As was mentioned in Section 3.3.4, many basic iterative schemes possess the smoothing property. Within a few iterations only, the highly oscillating error components can often be eliminated and

[3]Note that nonconvergent iterative schemes involving iteration matrices M with $\rho(M) \geq 1$ may be used as smoothers as well.

the smooth error modes remain. As a consequence, a coarser grid (i.e., a grid with fewer grid nodes) may still be sufficiently fine to represent this smooth error accurately enough. Note that, in general, it is the algebraic error (and not the approximation to the solution of the original linear system itself) that is smooth after a few steps of an appropriate basic iterative scheme have been performed.

The observation that the error is smooth after a few iterations motivates the idea to apply a few iterations of a suitable elementary iterative method on the respective fine grid. This step is called *pre-smoothing*. Then, an approximation to the remaining smooth error can be computed efficiently on a coarser grid, using a coarsened representation of (3.21); i.e., the residual equation. Afterwards, the smooth error must be interpolated back to the fine grid and, according to (3.20), added to the current fine grid approximation in order to correct the latter.

In the simplest case, the coarse grid is obtained by standard coarsening; i.e., by omitting every other row of fine-grid nodes in each dimension, see also Section 3.3.2. This coarsening strategy results in an equidistant coarse grid with mesh width $H = 2h$. As usual, we use Ω_h and Ω_H to represent the fine grid and the coarse grid, respectively. Furthermore, we assume that n_h and n_H denote the total numbers of unknowns corresponding to the fine grid and the coarse grid, respectively. Note that standard coarsening reduces the number of unknowns by a factor of approximately 2^{-d}, where d is the dimension of the problem [BHM00, TOS01].

The coarse representation of (3.21), which is used to approximate the current algebraic fine-grid error e_h, reads as

$$A_H e_H = r_H \ , \tag{3.22}$$

where $A_H \in \mathbf{R}^{n_H \times n_H}$ stands for the coarse-grid operator and $e_H, r_H \in \mathbf{R}^{n_H}$ are suitable coarse-grid representations of the algebraic fine-grid error e_h and the corresponding fine-grid residual r_h, respectively. Equation (3.22) must be solved for e_H.

3.3.5.2 Inter-Grid Transfer Operators

The combination of the fine-grid solution process and the coarse-grid solution process requires the definition of *inter-grid transfer operators*, which are necessary to map grid functions from the fine grid Ω_h to the coarse grid Ω_H, and vice versa. In particular, we need an *interpolation (prolongation) operator*

$$I_H^h \in \mathbf{R}^{n_h \times n_H} \ ,$$

which maps a coarse-grid function to a fine-grid function, as well as a *restriction operator*

$$I_h^H \in \mathbf{R}^{n_H \times n_h} \ ,$$

which maps a fine-grid function to a coarse-grid function. It is often desirable to define the restriction operator I_h^H to be a positive multiple of the transpose of the interpolation operator I_H^h; i.e.,

$$I_h^H = c \left(I_H^h\right)^T , \ \ c > 0 \ . \tag{3.23}$$

See the explanations below concerning the properties of the fine-grid operator A_h and the coarse-grid operator A_H.

The restriction operator I_h^H is used to transfer the fine-grid residual r_h to the coarse grid, yielding the right-hand side of the coarse-grid representation (3.22) of the fine-grid residual equation:

$$r_H := I_h^H r_h \ .$$

In the case of discrete operators A_h and A_H with slowly varying coefficients, typical choices for the restriction operator are *full weighting* or *half weighting*. These restriction operators compute weighted averages of the components of the fine-grid function to be restricted. They do not vary from grid point to grid point. In 2D, for example, they are given as follows:

- Full weighting:

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_h^H$$

- Half weighting:

$$\frac{1}{8}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}_h^H$$

Here, we have used the common stencil notation for restriction operators. The entries of these stencils specify the weights for the values of the respective grid function, when transferred from the fine grid Ω_h to the corresponding coarser grid Ω_H [BHM00, TOS01]. Representations of the full weighting and the half weighting operators in 3D are provided in [TOS01].

After the coarse representation e_H of the algebraic error has been determined, the interpolation operator is employed to transfer e_H back to the fine grid Ω_h:

$$\tilde{e}_h := I_H^h e_H \ .$$

We use $\tilde{e}_h$ to denote the resulting fine-grid vector, which is an approximation to the actual fine-grid error e_h. Ideally, the smoother would yield a fine-grid error e_h which lies in the range of I_H^h such that it could be eliminated completely by the correction $\tilde{e}_h$.

A typical choice for the prolongation operator is *linear interpolation*. In 2D, for example, this constant operator is given as follows:

$$\frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_H^h$$

Here, we have employed the common stencil notation for interpolation operators. Analogous to the above formalism for restriction operators, the entries of the interpolation stencils specify the weights for the values of the respective grid function, when prolongated from the coarse grid Ω_H to the corresponding finer grid Ω_h. Likewise, a representation of the linear interpolation operator in 3D is provided in [TOS01]. Note that the inter-grid transfer operators *full weighting* and *linear interpolation* satisfy (3.23) with $c = \frac{1}{4}$.

If the stencil coefficients vary significantly from grid node to grid node, it may be necessary to employ operator-dependent inter-grid transfer operators, which do not just compute weighted averages when mapping fine-grid functions to the coarse grid, and vice versa [ABDP81, BHM00]. See also [Bra84] for a discussion of how to select appropriate restriction and prolongation operators depending on the order of the differential operator L in (3.16).

Ideally, the high-frequency components dominate the fine-grid error after the coarse-grid representation e_H of the error has been interpolated to the fine grid Ω_h and added to the current fine-grid approximation u_h; i.e., after the *correction*

$$u_h \leftarrow u_h + \tilde{e}_h$$

has been carried out. In addition, the interpolation of the coarse-grid approximation e_H to the fine grid Ω_h usually even amplifies oscillatory error components. Therefore, a few further iterations of the smoother are typically applied to the fine-grid solution u_h. This final step is called *post-smoothing*.

3.3.5.3 Coarse-Grid Operators

The coarse-grid operator A_H can be obtained by discretizing the continuous differential operator L from (3.16) on the coarse grid Ω_H anew. Alternatively, A_H can be computed as the so-called *Galerkin product*

$$A_H := I_h^H A_h I_H^h \ . \tag{3.24}$$

The construction of the coarse-grid operator A_H using (3.24) is primarily motivated algebraically. An immediate observation is the following. If the fine-grid operator A_h as well as the inter-grid transfer operators I_h^H and I_H^h are characterized by *compact* stencils (i.e., 3-point stencils in 1D, 9-point stencils in 2D, or 27-point stencils in 3D) the resulting coarse-grid operator A_H will be given by corresponding compact stencils as well. In a multigrid implementation, this property enables the use of simple data structures on all levels of the grid hierarchy, cf. Section 8.4.

Note that, if both (3.23) and (3.24) hold, a *symmetric* fine-grid operator A_h yields a symmetric coarse-grid operator A_H, because

$$(A_H)^T = \left(I_h^H A_h I_H^h\right)^T = \left(I_H^h\right)^T (A_h)^T \left(I_h^H\right)^T = c^{-1} I_h^H A_h c I_H^h = A_H \ .$$

If the fine-grid operator A_h is even *symmetric positive definite* (i.e., if we have $A_h = A_h^T$ and $v_h^T A_h v_h > 0$ for all $v_h \in \mathbf{R}^{n_h}$, $v_h \neq 0$) and the interpolation operator I_H^h has full rank (i.e., $\ker I_H^h = \{0\}$), the corresponding coarse-grid operator A_H will again be symmetric positive definite. The positive definiteness of A_H can be seen as follows. Assume $v_H \in \mathbf{R}^{n_H}$ with $v_H \neq 0$. Then, we find that

$$v_H^T A_H v_H = v_H^T \left(I_h^H A_h I_H^h\right) v_H = v_H^T c \left(I_H^h\right)^T A_h I_H^h v_H = c \left(I_H^h v_H\right)^T A_h \left(I_H^h v_H\right) > 0 \ .$$

Here, we have used that $c > 0$, that $v_H \neq 0$ implies $I_H^h v_H \neq 0$ (since $\ker I_H^h = \{0\}$), as well as the positive definiteness of the fine-grid operator A_h itself.

One important consequence of this observation is based on the result that symmetric positive definite matrices are regular [GL98, Hac93]. Note that the definition of a positive definite matrix A in [Hac93] assumes A to be symmetric. However, the regularity of A follows immediately from its positive definiteness alone; i.e., from the fact that $v_h^T A_h v_h > 0$ for $v_h \neq 0$.

As a consequence, if the matrix A_h corresponding to the finest grid of the hierarchy is symmetric positive definite and, furthermore, both the inter-grid transfer operators and the coarse-grid matrices are chosen appropriately, each of the coarse-grid matrices will be symmetric positive definite and thus regular as well. This property of the hierarchy of discrete operators often simplifies the analysis of multigrid methods. See also Sections 8.3 and 8.4.

3.3.5.4 Two-Grid Method

Based on the previous considerations, we can now formulate the two-grid *coarse-grid correction (CGC)* V(ν_1,ν_2)-cycle. Algorithm 3.2 demonstrates a single iteration of this method. The scheme assumes that, in each iteration, the coarse-grid equation is solved exactly. This can be seen from the fact that the inverse of the coarse-grid operator A_H is used in Step 4. In practice, the exact

solution of the coarse-grid equation can be determined by applying a direct or an iterative solver, see Section 3.2.

The notation $S_h^{\nu}(\cdot)$ is introduced to indicate that ν iterations of an appropriate smoothing method are applied to the corresponding approximation on Ω_h. Hence, the two parameters ν_1 and ν_2 ($\nu_1, \nu_2 \geq 0$) specify the numbers of pre-smoothing and post-smoothing iterations, respectively. Common values for ν_1 and ν_2 are 1, 2, or 3. We assume that the initial guess $u_h^{(0)}$, the right-hand side f_h on the fine grid Ω_h, as well as the matrices A_h and A_H have been initialized beforehand.

Figure 3.2 once more illustrates a single iteration of a two-grid V(ν_1,ν_2)-cycle correction scheme, computing $u_h^{(k+1)}$ from $u_h^{(k)}$.

Note that the two-grid V(ν_1,ν_2)-cycle correction scheme represents another linear iterative method. It can be written in the form of (3.4) — see Section 3.2.1.3 — as follows. First, it is easy to see that the iteration matrix $\bar{M}_h^H$ of the coarse-grid correction operator alone can be written as

$$\bar{M}_h^H := I - I_H^h A_H^{-1} I_h^H A_h \ , \tag{3.25}$$

where the corresponding approximate inverse $\bar{N}_h^H$ of A_h (i.e., $\bar{N}_h^H \approx A_h^{-1}$) is given by

$$\bar{N}_h^H := I_H^h A_H^{-1} I_h^H \ ,$$

cf. (3.5). Then, based on the representation given in (3.25), we obtain that the iteration matrix M_h^H of the two-grid method is given by the product

$$M_h^H := \bar{S}_h^{\nu_2} \bar{M}_h^H \bar{S}_h^{\nu_1} \ , \tag{3.26}$$

where $\bar{S}$ stands for the iteration matrix of the smoother. See [ST82, TOS01] for a convergence analysis of the two-grid method based on this representation.

3.3.5.5 Generalization to Multigrid

In the course of each two-grid V-cycle, the linear system corresponding to the coarse grid must be solved exactly, cf. Algorithm 3.2, Step 4. If this linear system is still too large to be solved efficiently by using either a direct method or an elementary iterative method, the idea of applying a coarse grid correction (i.e., the idea of solving the corresponding residual equation on a coarser grid) can be applied recursively. This leads to the class of *multigrid* schemes.

Algorithm 3.3 shows the structure of a single multigrid V(ν_1,ν_2)-cycle. Due to its recursive formulation, it is sufficient to distinguish between a fine grid Ω_h and a coarse grid Ω_H. When the recursive scheme calls itself in Step 7, the current coarse grid Ω_H becomes the fine grid of the next deeper invocation of the multigrid V-cycle procedure. Typically, $u_H := 0$ is used as initial guess on Ω_H.

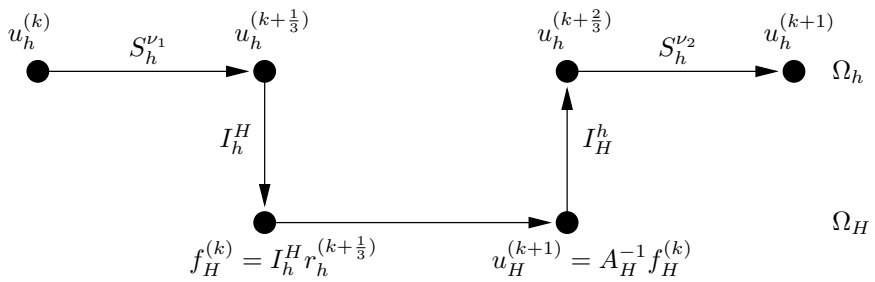


Figure 3.2: Two-grid CGC V(ν_1,ν_2)-cycle.

Algorithm 3.2 Two-grid CGC V(ν_1,ν_2)-cycle.

1: Perform ν_1 iterations of the smoother on Ω_h (pre-smoothing):

$$u_h^{(k+\frac{1}{3})} \leftarrow S_h^{\nu_1}\left(u_h^{(k)}\right)$$

2: Compute the residual on Ω_h:

$$r_h \leftarrow f_h - A_h u_h^{(k+\frac{1}{3})}$$

3: Restrict the residual from Ω_h to Ω_H:

$$f_H \leftarrow I_h^H r_h$$

4: Solve the coarse-grid equation on Ω_H:

$$u_H \leftarrow A_H^{-1} f_H$$

5: Interpolate the coarse-grid solution (i.e., the error) from Ω_H to Ω_h:

$$\tilde{e}_h \leftarrow I_H^h u_H$$

6: Correct the fine-grid approximation on Ω_h:

$$u_h^{(k+\frac{2}{3})} \leftarrow u_h^{(k+\frac{1}{3})} + \tilde{e}_h$$

7: Perform ν_2 iterations of the smoother on Ω_h (post-smoothing):

$$u_h^{(k+1)} \leftarrow S_h^{\nu_2}\left(u_h^{(k+\frac{2}{3})}\right)$$

Analogous to the case of the two-grid method, the parameters ν_1 and ν_2 denote the numbers of iterations of the smoother before and after the coarse-grid correction, respectively. Likewise, typical values for ν_1 and ν_2 are 1, 2, or 3. Again, we assume that the initial guess $u_h^{(0)}$ and the right-hand side f_h on the very finest grid, as well as all matrices corresponding to the individual grid levels of the hierarchy have been initialized beforehand. Note that all cache-optimized multigrid codes we will analyze in Section 7.3 implement CGC V(ν_1,ν_2)-cycles.

An additional parameter γ may be introduced in order to increase the number of multigrid cycles to be executed on the coarse grid Ω_H in Step 7 of Algorithm 3.3. This parameter γ is called the *cycle index*. The choice $\gamma := 1$ (as is implicitly the case in Algorithm 3.3) leads to multigrid V(ν_1,ν_2)-cycles, while different cycling strategies are possible. Another common choice is $\gamma := 2$, which leads to the multigrid W-cycle [BHM00, TOS01]. The names of these schemes are motivated by the order in which the various grid levels are visited during the multigrid iterations. We will return to the issue of cycling strategies in Section 8.3, where we will present a different interpretation of the multigrid principle.

Figure 3.3 shows the algorithmic structure of a three-grid CGC V(ν_1,ν_2)-cycle. This figure particularly illustrates the origin of the term V-cycle. We have used the level indices h, $2h$, and $4h$ in order to indicate that we generally assume the case of standard coarsening, see Sections 3.3.2 and 3.3.5.1.

As is the case for the two-grid iteration, the multigrid V(ν_1,ν_2)-cycle correction scheme shown in Algorithm 3.3 represents another linear iterative method which can be written in the form of

Algorithm 3.3 Recursive definition of the multigrid CGC V(ν_1,ν_2)-cycle.

1: Perform ν_1 iterations of the smoother on Ω_h (pre-smoothing):

$$u_h^{(k+\frac{1}{3})} \leftarrow S_h^{\nu_1}\left(u_h^{(k)}\right)$$

2: Compute the residual on Ω_h:

$$r_h \leftarrow f_h - A_h u_h^{(k+\frac{1}{3})}$$

3: Restrict the residual from Ω_h to Ω_H and initialize the coarse-grid approximation:

$$f_H \leftarrow I_h^H r_h \ , \ u_H \leftarrow 0$$

4: **if** Ω_H is the coarsest grid of the hierarchy **then**

5: Solve the coarse-grid equation $A_H u_H = f_H$ on Ω_H exactly

6: **else**

7: Solve the coarse-grid equation $A_H u_H = f_H$ on Ω_H approximately by (recursively) performing a multigrid V(ν_1,ν_2)-cycle starting on Ω_H

8: **end if**

9: Interpolate the coarse-grid approximation (i.e., the error) from Ω_H to Ω_h:

$$\tilde{e}_h \leftarrow I_H^h u_H$$

10: Correct the fine-grid approximation on Ω_h:

$$u_h^{(k+\frac{2}{3})} \leftarrow u_h^{(k+\frac{1}{3})} + \tilde{e}_h$$

11: Perform ν_2 iterations of the smoother on Ω_h (post-smoothing):

$$u_h^{(k+1)} \leftarrow S_h^{\nu_2}\left(u_h^{(k+\frac{2}{3})}\right)$$

(3.4). Again, the derivation of the corresponding iteration matrix which is based on the recursive application of the involved two-grid iteration matrices — see (3.26) in the previous section — can be found in [ST82].

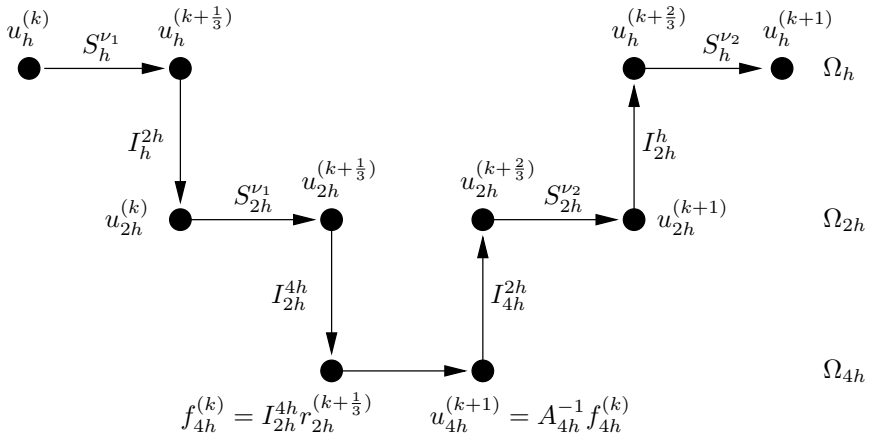


Figure 3.3: Three-grid CGC V(ν_1,ν_2)-cycle.

3.3.5.6 Remarks on Multigrid Convergence Analysis

A common and powerful approach towards the convergence analysis (and the development) of multigrid methods is based on *local Fourier analysis (LFA)*. The principle of the LFA is to examine the impact of the discrete operators, which are involved in the two-grid or in the multigrid setting, by representing them in the basis of the corresponding Fourier spaces. For simplicity reasons, the LFA ignores boundary conditions and, instead, assumes infinite grids [Bra77, TOS01].

Alternatively, the convergence analysis of the two-grid scheme and the multigrid scheme can be based on the notions of *smoothing property* and *approximation property*, which have been introduced by Hackbusch [Hac85]. See also [Hac93]. As these names suggest, the smoothing property states that the smoother eliminates high-frequency error components without introducing smooth ones. In contrast, the approximation property states that the CGC performs efficiently; i.e., that the inverse of the coarse-grid operator represents a reasonable approximation to the inverse of the fine-grid operator.

3.3.5.7 Full Approximation Scheme

So far, we have considered the CGC scheme, or simply the *correction scheme (CS)*. This means that, in each multigrid iteration, any coarse grid is employed to compute an approximation to the error on the next finer grid.

Alternatively, the coarse grid can be used to compute an approximation to the fine-grid solution instead. This approach leads to the *full approximation scheme/storage (FAS) method*, see [Bra77, BHM00, TOS01]. The FAS method is primarily used if the discrete operator is nonlinear or if adaptive grid refinement is introduced [Bra97, KA00]. In the latter case, the finer grids may not cover the entire domain in order to reduce both memory consumption and computational work [Bra77, Rüd93b, TOS01].

In the following, we will briefly describe a two-grid FAS V-cycle. As usual, we use Ω_h and Ω_H to refer to the fine grid and to the coarse grid, respectively. We assume that the fine-grid equation takes the form

$$N_h u_h = f_h \ . \tag{3.27}$$

We denote the discrete fine-grid operator as N_h, $N_h : \mathbf{R}^{n_h} \to \mathbf{R}^{n_h}$, in order to indicate that it might be nonlinear; i.e., (3.27) might represent a large system of nonlinear algebraic equations. Again, we use n_h and n_H to denote the numbers of unknowns on Ω_h and on Ω_H, respectively. Using the notation introduced in Section 3.3.3, $u_h^* \in \mathbf{R}^{n_h}$ explicitly denotes the exact fine-grid solution, and $f_h \in \mathbf{R}^{n_h}$ is the corresponding right-hand side; i.e., we have

$$N_h u_h^* = f_h \ . \tag{3.28}$$

Furthermore, $u_h \in \mathbf{R}^{n_h}$ stands for an approximation to u_h^*, and the algebraic error $e_h \in \mathbf{R}^{n_h}$ is defined as in (3.20). Finally, the fine-grid residual $r_h \in \mathbf{R}^{n_h}$ is given by

$$r_h = f_h - N_h u_h \ . \tag{3.29}$$

The fundamental observation is that, in the case of a nonlinear discrete operator N_h, the residual equation cannot be written in the form of (3.21). Instead, subtracting (3.28) from (3.29) only yields

$$N_h u_h^* - N_h u_h = r_h \ , \tag{3.30}$$

which cannot be simplified further. Note that, if N_h is linear, (3.30) will be equivalent to (3.21).

The fundamental idea behind the CGC scheme and the FAS method is the same; after a few iterations of a suitable smoother on the fine grid Ω_h, the resulting algebraic error e_h is smooth (i.e., its high-frequency components have been reduced sufficiently) and can be approximated efficiently on the coarser grid Ω_H. In the case of a nonlinear operator N_h, an appropriate nonlinear smoothing method (e.g., nonlinear Gauss-Seidel iteration) must be applied [BHM00, KA00, TOS01].

We introduce N_H, $N_H : \mathbf{R}^{n_H} \rightarrow \mathbf{R}^{n_H}$, to denote an appropriate representation of N_h on the coarse grid Ω_H. We further assume $\hat{u}_H \in \mathbf{R}^{n_H}$, $e_H \in \mathbf{R}^{n_H}$, and $r_H \in \mathbf{R}^{n_H}$ to be coarse-grid representations of u_h, e_h, and r_h, respectively, where $\hat{u}_H$ and r_H are obtained by restricting u_h and r_h from the fine grid Ω_h to the coarse grid Ω_H:

$$\hat{u}_H := \hat{I}_h^H u_h \ ,$$

and

$$r_H := I_h^H r_h \ .$$

We have used the symbols $\hat{I}_h^H$ and I_h^H to indicate that different restriction operators could be applied here. A common choice for $\hat{I}_h^H$ is injection [TOS01].

Using the previous notation, the coarse-grid version of (3.30) reads as

$$N_H u_H = N_H \hat{u}_H + r_H \ , \tag{3.31}$$

where

$$u_H := \hat{u}_H + e_H \ .$$

Equation (3.31) is solved for u_H and, afterwards, the coarse-grid error approximation $e_H = u_H - \hat{u}_H$ is interpolated back to the fine grid Ω_h:

$$\tilde{e}_h := I_H^h e_H \ .$$

As is the case for the multigrid CGC scheme, we actually employ the coarse-grid equation (3.31) in the FAS context in order to determine a coarse-grid approximation e_H to the (smooth) fine-grid error e_h. Again, in contrast to the fine-grid error e_h, the fine-grid approximation u_h is generally not smooth at all and, therefore, cannot be represented accurately on the coarse grid Ω_H.

As for the case of the multigrid CGC scheme, the interpolated error $\tilde{e}_h$ is used to correct the current fine-grid approximation u_h. Finally, a few iterations of the smoothing method are commonly applied to eliminate high-frequency error components which dominate the fine-grid approximation u_h after the correction step has been performed, cf. Section 3.3.5.2.

The extension of the two-grid FAS V-cycle to the multigrid FAS V-cycle parallels the extension of the two-grid CGC method. Analogous to Algorithm 3.3, Algorithm 3.4 illustrates the structure of a single multigrid FAS V(ν_1,ν_2)-cycle. Due to its recursive formulation, it is again sufficient to distinguish between a fine grid Ω_h and a coarse grid Ω_H. Likewise, $u_H := 0$ represents the usual initial guess on Ω_H. Analogously, a cycle index γ can further be introduced into Step 8 of Algorithm 3.4 in order to increase the number of multigrid FAS cycles to be performed on Ω_H and, therefore, to control the cycling strategy.

Introducing the coarse-grid vectors

$$f_H := I_h^H f_h \in \mathbf{R}^{n_H}$$

as well as

$$\tau_h^H(u_h) := N_H \hat{I}_h^H u_h - I_h^H N_h u_h \in \mathbf{R}^{n_H} \ ,$$

Algorithm 3.4 Recursive definition of the multigrid FAS V(ν_1,ν_2)-cycle.

1: Perform ν_1 iterations of the smoother on Ω_h (pre-smoothing):

$$u_h^{(k+\frac{1}{3})} \leftarrow S_h^{\nu_1}\left(u_h^{(k)}\right)$$

2: Compute the residual on Ω_h:

$$r_h \leftarrow f_h - N_h u_h^{(k+\frac{1}{3})}$$

3: Restrict the fine-grid approximation from Ω_h to Ω_H:

$$\hat{u}_H \leftarrow \hat{I}_h^H u_h^{(k+\frac{1}{3})}$$

4: Restrict the residual from Ω_h to Ω_H and initialize the coarse-grid approximation:

$$r_H \leftarrow I_h^H r_h \ , \ u_H \leftarrow 0$$

5: **if** Ω_H is the coarsest grid of the hierarchy **then**

6: Solve the coarse-grid equation $N_H u_H = N_H \hat{u}_H + r_H$ on Ω_H exactly

7: **else**

8: Solve the coarse-grid equation $N_H u_H = N_H \hat{u}_H + r_H$ on Ω_H approximately by (recursively) performing a multigrid FAS V(ν_1,ν_2)-cycle starting on Ω_H

9: **end if**

10: Compute the coarse-grid correction:

$$e_H \leftarrow u_H - \hat{u}_H$$

11: Interpolate the coarse-grid correction (i.e., the error) from Ω_H to Ω_h:

$$\tilde{e}_h \leftarrow I_H^h e_H$$

12: Correct the fine-grid approximation on Ω_h:

$$u_h^{(k+\frac{2}{3})} \leftarrow u_h^{(k+\frac{1}{3})} + \tilde{e}_h$$

13: Perform ν_2 iterations of the smoother on Ω_h (post-smoothing):

$$u_h^{(k+1)} \leftarrow S_h^{\nu_2}\left(u_h^{(k+\frac{2}{3})}\right)$$

we find that (3.31) can be rewritten in the following form:

$$\begin{aligned} N_H u_H &= N_H \hat{u}_H + r_H \\ &= N_H \hat{u}_H + I_h^H (f_h - N_h u_h) \\ &= f_H + \left(N_H \hat{I}_h^H u_h - I_h^H N_h u_h\right) \\ &= f_H + \tau_h^H(u_h) \ . \end{aligned}$$

This emphasizes that the coarse-grid equation in an FAS setting does not simply correspond to the original continuous problem — cf. (3.16) and (3.17) — being discretized anew on the coarse grid Ω_H. Rather, the term $\tau_h^H(u_h)$, which is called the (h,H)-*relative truncation error*, can be seen as a correction to the right-hand side f_H on the coarse grid Ω_H in order to enhance the accuracy of

the coarse-grid solution [Bra77, BHM00]. Clearly, if $u_h = u_h^*$ holds, it follows that

$$N_H u_H = N_H \hat{I}_h^H u_h^* \ ,$$

which means that the exact solution u_H of the coarse-grid problem coincides with the restricted fine-grid solution $\hat{I}_h^H u_h^*$, as long as N_H is injective.

Note that the quantity $\tau_h^H(u_h)$ plays a fundamental role in the context of extrapolation techniques for multigrid algorithms [Ber97, Hac85] and in the case of adaptive mesh refinement [TOS01]. We will return to the FAS method in Section 8.3 in the context of the fully adaptive multigrid method.

3.3.5.8 Nested Iteration and Full Multigrid

In most cases, the solution times of iterative methods (i.e., the numbers of iterations required to fulfill the given stopping criteria) can be reduced drastically by choosing suitable initial guesses. When applied recursively, the idea of determining an approximation on a coarse grid first and interpolating this approximation afterwards in order to generate an accurate initial guess on a fine grid leads to the principle of *nested iteration* [BHM00, Hac93].

The combination of nested iteration and the multigrid schemes we have described so far further leads to the class of *full multigrid (FMG) methods*, which typically represent the most efficient multigrid algorithms. This means that full multigrid schemes are asymptotically optimal; i.e., for typical problems, the computational work required to solve the discrete problem on the finest grid level up to discretization accuracy grows only linearly with the number of unknowns [Bra84, TOS01].

The FMG scheme generally starts on the coarsest level of the grid hierarchy. There, an approximation to the solution is computed and then interpolated to the next finer grid, yielding a suitable initial guess on this next finer grid. A certain number of multigrid cycles (either CGC-based or FAS-based) is applied to improve the approximation, before it is in turn interpolated to the next finer grid, and so on.

As Algorithm 3.5 shows, the FMG scheme can be formulated recursively. The linear system on the coarsest level of the grid hierarchy is assumed to be solved exactly. Since the approximations

Algorithm 3.5 Recursive formulation of the FMG scheme on Ω_h.

1: **if** Ω_h is the coarsest grid of the hierarchy **then**
2: Solve $A_h u_h = f_h$ on Ω_h exactly
3: **else**
4: Restrict the right-hand side from Ω_h to the next coarser grid Ω_H:

$$f_H \leftarrow I_h^H f_h$$

5: Solve $A_H u_H = f_H$ using FMG on Ω_H recursively
6: Interpolate the coarse-grid approximation from Ω_H to Ω_h in order to obtain a good initial guess on Ω_h:

$$u_h^{(0)} \leftarrow \tilde{I}_H^h u_H$$

7: Improve the approximation on Ω_h by applying ν_0 multigrid iterations:

$$u_h \leftarrow \mathrm{MG}_{\nu_1,\nu_2}^{\nu_0}\left(u_h^{(0)}, A_h, f_h\right)$$

8: **end if**

which are mapped from coarse to fine grids in Step 6 are not necessarily smooth and potentially large, it is commonly recommended to choose an interpolation operator $\tilde{I}_H^h$ of sufficiently high order. More precisely, based on practical experience, it is recommended to choose an interpolation operator $\tilde{I}_H^h$ whose order is higher than the order of the discretization error (i.e., the error that results from the transformation of the continuous problem into a discrete version), see [TOS01] for details.

Depending on the actual problem, the multigrid method applied in Step 7 of Algorithm 3.5 may either be based on the CGC scheme or on the FAS method. It may involve either V(ν_1,ν_2)-cycles or W(ν_1,ν_2)-cycles. The notation we have introduced in Step 7 is supposed to indicate that ν_0 multigrid cycles are performed on the linear system involving the matrix A_h and the right-hand side f_h, starting with the initial guess $u_h^{(0)}$ which has been determined previously by interpolation in Step 6.

Note that, in order to compute an approximation to the actual solution on each of the coarse grids, it is necessary to appropriately represent the original right-hand side; i.e., the right-hand side on the finest grid level. These coarse-grid right-hand sides are needed whenever the corresponding coarse-grid level is visited for the first time and, in the case of the FAS method, during the multigrid cycles as well. They can either be determined by successively restricting the right-hand side from the finest grid (as is the case in Algorithm 3.5, Step 4) or, alternatively, by discretizing the continuous right-hand side (i.e., the function f in (3.16)) on each grid level anew [BHM00].

As an alternative to multigrid V-cycles, our patch-adaptive multigrid codes implement the FMG scheme as well. Both the CGC scheme and the FAS method can be used as multigrid plug-ins (cf. Algorithm 3.5, Step 7) in the FMG framework, see Section 8.4.

Chapter 4

Numerical Algorithms II — Cellular Automata

In this chapter, we will first briefly review the definition of a *cellular automaton (CA)* and then focus on a special class of CA[1]; the *lattice Boltzmann automata* which are used in the *lattice Boltzmann method*.

4.1 Formal Specification of CA

CA represent an (intrinsically parallel) approach to model and to simulate the time-dependent behavior of complex systems. CA were first studied by J. v. Neumann at the end of the 1940s. His motivation was to formalize and to investigate the mechanism of self-reproduction; e.g., in biological systems. One of today's most popular examples of CA is J. Conway's *game of life* [Gut91].

According to [CD98b], a CA formally consists of

1. a regular *grid (lattice)* of *cells (lattice sites)* covering a portion of the d-dimensional space (typically $\mathbf{Z}^d$ in the discrete case or $\mathbf{R}^d$ in the continuous case),

2. a set $S(x,t) := \{S_1(x,t), S_2(x,t), \ldots, S_m(x,t)\}$ of Boolean variables, each depending on both the position x of the cell[2] and the current discrete time t, as well as

3. a (local) *rule (state transition function)* $R := \{R_1, R_2, \ldots, R_m\}$ which specifies the time evolution of the states $S(x,t)$ as follows:

$$S_i(x, t+1) := R_i \left(S(x,t), S(x+\delta_1, t), S(x+\delta_2, t), \ldots, S(x+\delta_q, t) \right) \quad , \tag{4.1}$$

   where $1 \leq i \leq m$ and the cell positions $x + \delta_j$, $1 \leq j \leq q$, define the *neighborhood* of the cell at position x. Note that, according to this definition, each cell can be in one of a finite number of states. For the introduction of probabilistic (nondeterministic) behavior into this formalism, we refer once more to [CD98b].

The neighborhood of the cell at position x is the set of all cells which need to be accessed in order to compute the next state $S(x, t+1)$ of that cell. In 2D, typical neighborhoods are the cell

[1] We use the abbreviation *CA* for the singular and for the plural case.

[2] In the case of the continuous d-dimensional space, the *position* $x \in \mathbf{R}^d$ of the cell is usually defined by the Cartesian coordinates of its center point.

itself and its four geographical neighbors north, east, south, and west *(von Neumann neighborhood)*, as well as the cell itself and its eight surrounding cells *(Moore neighborhood)*.

In comparison with discretization techniques for PDEs on regular 2D grids, the von Neumann neighborhood thus corresponds to a 5-point stencil, whereas the Moore neighborhood corresponds to a 9-point stencil. It is exactly this *local access pattern* which both CA and the iterative algorithms that we consider have in common, and which allows for the application of our data locality optimizations. We will discuss these optimizing transformations in Chapters 5 and 6.

Furthermore, if we compare the update rule for CA given by (4.1) with the update scheme of Jacobi's method, cf. (3.12) in Section 3.2.2, we observe that the data access patterns parallel each other. In order to compute the new state $S(x, t+1)$ of the cell at position x at time $t+1$, we must read the states $S(x+\delta_j, t)$ of the cells in its neighborhood at time t. Correspondingly, in order to compute a new approximation $u_i^{(k+1)}$ for the i-th component of the iterate $u^{(k+1)}$ using Jacobi's method, we must access the previous approximations $u_j^{(k)}$ of all components j on which the i-th unknown depends. We will return to this correspondence in Section 5.4, where a general data layout optimization approach for CA and Jacobi-type algorithms will be introduced.

Note that the rule R in the definition of a CA is *homogeneous*; i.e., it does not depend on the position x of the cell to be updated. This means that, in this formalism, spatial and temporal inhomogeneities must be represented by suitably chosen state variables $S_i(x, t)$. In particular, it assumes an infinite lattice and, hence, *boundary cells* surrounding the actual finite computational domain must be introduced in this manner, see [CD98b].

The CA approach is not necessarily a rigid formalism, but rather a philosophy of modeling which allows for various generalizations. The extension to each cell being in one of an infinite instead of a finite number of states is rather straightforward. For example, each state variable $S_i(x, t)$ can be defined to be an integer or a real variable instead of a Boolean variable. The lattice Boltzmann method, for example, represents such a generalization of the classical framework, see Section 4.2. It maintains the discretizations in space and time, but each lattice site can be in one of an infinite set of states. This infinite set of states is given by a (finite) set of real numbers representing distribution functions[3].

There is a large body of references in the field of CA. Introductions to the theory of CA with special emphasis on the modeling and simulation of physical systems are provided in [CD98b, TM87], for example. We particularly refer to the illustrative introduction to CA presented in [Fla98]. A rich collection of articles concerning theory and application of CA can be found in [Gut91]. In addition to classical applications in physics and chemistry, these articles also cover more computer science oriented topics, such as computation-theoretic aspects and neural networks. [Wol94] provides a collection of papers on CA by S. Wolfram, many of which also deal with topics in the area of theoretical computer science; e.g., cryptography and complexity theory. Aspects of CA-based parallel high performance simulation are discussed in [Tal00]. This article also contains a list of references to various practical applications of CA; e.g., automobile traffic flow, pattern recognition, and VLSI simulation. Cache performance aspects of parallel implementations of CA are discussed in [DAC00], see also Section 9.1.3.4.

[3]Typical implementations of the LBM, however, use floating-point numbers to represent these real quantities. Consequently, the state space of each variable $S_i(x, t)$ is then again finite due to the finite set of machine-representable numbers [Ueb97a].

4.2 The Lattice Boltzmann Method

4.2.1 Introduction

The applicability of CA to model and to simulate dynamic systems in physics was not recognized before the 1980s. Research in this field has led to the theory of the *lattice-gas cellular automata (LGCA)* and to the *lattice Boltzmann method (LBM)*, on which we will focus in the following. We provide only a brief description of the LBM in this section, since the discussion of the actual physics behind this approach is beyond the scope of this work and not essential for the application of our optimization techniques. Instead, we refer the reader to the literature; e.g., [CD98a, CD98b, WG00]. In particular, [CDE94] contains an easy-to-read introduction to the LBM. The notation we use in this thesis is based on [WG00].

The usual approach towards solving problems in computational fluid dynamics is based on the numerical solution of the governing partial differential equations, particularly the Navier-Stokes equations. The idea behind this approach is to discretize the computational domain using finite differences, finite elements, or finite volumes, to derive algebraic systems of equations, and to solve these systems numerically [GDN98]. This can be seen as a top-down approach which starts from a macroscopic description of the dynamic behavior of the system; i.e., the governing PDEs.

In contrast, the LBM represents a bottom-up approach. It is a particle-oriented technique, which is based on a relatively simple microscopic model representing the kinetics of the moving fluid particles. Using x to denote the position in space, u to denote the particle velocity, and t to denote the time parameter, the *particle distribution function* $f(x, u, t)$ is discretized in space, velocity, and time. This results in the computational domain being regularly divided into grid cells (lattice sites). In each time step, particles "jump" to neighboring lattice sites and then scatter.

The current state of each lattice site is defined by an array of floating-point numbers which represent the distribution functions with respect to the discrete directions of velocity. In contrast to the *lattice gas* approach where hexagonal grids are more common, we only focus on orthogonal grids in 2D and 3D, since they are almost exclusively used for the LBM [WG00][4]. These grids consist of squares and cubes, respectively.

The LBM proceeds as follows. In each time step, the entire grid is traversed, and the distribution function values at each site are updated according to the states of itself and its neighboring sites in the grid at the previous discrete point in time. From this, it becomes clear that the LBM is a canonical extension of the formalism of cellular automata, which we have introduced previously in Section 4.1. The update step in the LBM consists of a *stream* operation, where the corresponding data from the neighboring sites is retrieved, and a *collide* operation, where the new distribution function values at the current site are computed according to the underlying model of the microscopic behavior of the fluid particles, preserving hydrodynamic quantities such as mass density, momentum density, and energy.

Note that, in every time step, the lattice may be traversed in any order since the LBM only accesses data corresponding to the previous point in time in order to compute new distribution function values. We have already mentioned in Section 4.1 that, from an abstract point of view, the structure of the LBM thus parallels the structure of an implementation of Jacobi's method for the iterative solution of linear systems on structured meshes. In particular, the time loop in the LBM corresponds to the iteration loop in Jacobi's method.

[4] Due to their superior stability properties, more complex lattices can for example be used in the case of *thermal lattice Boltzmann models* [VPVM98].

In this thesis, we will exclusively focus on the LBM as an example of a widely used class of CA. As was mentioned in Section 4.1, it is the locality of the data access patterns which CA generally exhibit and which enables the application of our optimizing transformations. Therefore, the actual application of the CA is irrelevant as long as the performance of its implementations on cache-based architectures — in terms of cell updates per time unit — are memory-bound.

4.2.2 Discretization of the Boltzmann Equation

Usually, the underlying kinetic model of the LBM is derived from the *Boltzmann equation*

$$\frac{\partial f}{\partial t} + \langle u, \nabla f \rangle = Q \tag{4.2}$$

which describes the time-dependent behavior of the *particle distribution function* $f = f(x, u, t)$. In this equation, Q represents the *collision operator*, $\langle \cdot, \cdot \rangle$ denotes the standard inner product, and ∇f denotes the gradient of f with respect to the spatial dimensions. In order to simplify the model, we assume a particular collision operator Q; the *BGK approximation (single relaxation time approximation)* [HL97, WG00]:

$$Q = -\frac{1}{\tau}\left(f - f^{(0)}\right)$$

Here, τ is the *relaxation time*, and $f^{(0)} = f^{(0)}(x, u, t)$ denotes the *equilibrium distribution function (Boltzmann-Maxwellian distribution function)*. This simplification of the collision operator basically means that the collisions of the particles are not defined explicitly. Instead, they are mathematically represented as transitions towards the local equilibrium states. Hence, (4.2) becomes

$$\frac{\partial f}{\partial t} + \langle u, \nabla f \rangle = -\frac{1}{\tau}\left(f - f^{(0)}\right) \quad . \tag{4.3}$$

As we have already mentioned previously, this continuous model is now discretized in space, velocity, and time. The introduction of a finite set $\{v_i\}$, $0 \leq i \leq n$, of velocities implies that we only have to deal with a remaining finite number of associated distribution functions $f_i = f_i(x, t) := f(x, v_i, t)$.

Consequently, the *discrete Boltzmann equation*:

$$\frac{\partial f_i}{\partial t} + \langle v_i, \nabla f_i \rangle = -\frac{1}{\tau}\left(f_i - f_i^{(0)}\right) \tag{4.4}$$

can be derived from (4.3).

Equation (4.4) is then nondimensionalized, see [WG00] for the corresponding technical details. In particular, this means that the discrete velocities v_i are replaced by the *lattice velocities* $c_i := v_i/U$, where U is the reference velocity, and the distribution functions $f_i(x, t)$ are replaced by $F_i(x, t) := f_i(x, t)/\eta_r$, where η_r is the reference density.

Next, we apply a first-order difference scheme to approximate the time derivative $\partial F_i(x, t)/\partial t$ and first-order difference schemes to approximate the derivatives along the spatial dimensions which occur in the gradient $\nabla F_i(x, t)$. We assume that the grid spacing Δx, the time step Δt, and the lattice velocities c_i are chosen appropriately; i.e., $\Delta x/\Delta t = c_i$. This means that each lattice velocity is defined such that, in each time step, the corresponding particles cover exactly the distance prescribed by the grid spacing; i.e., by the size of the lattice cells.

Finally, we obtain the *(BGK) lattice Boltzmann equation* [WG00]:

$$F_i(x + c_i\Delta t, t + \Delta t) - F_i(x, t) = -\frac{1}{\tau}\left(F_i(x, t) - F_i^{(0)}(x, t)\right) \quad . \tag{4.5}$$

This is the equation on which all our implementations of the LBM are based on. It states that the i-th distribution function of the cell at position $x + c_i\Delta t$ at time $t + \Delta t$ equals the i-th distribution function at the corresponding neighboring cell at position x at time t plus some contribution resulting from the previous collision of the particles in cell x at time t. This equation therefore describes how the information is propagated discretely in time and space.

In the LBM, it is necessary to compute the *mass density* $\rho(x,t)$ and the *momentum density* $j(x,t)$ in each time step, because they are needed during the calculation of the equilibrium distribution functions $F_i^{(0)}(x,t)$. These quantities are defined as follows:

$$\rho(x,t) := \sum_i F_i(x,t) \ , \tag{4.6}$$

$$j(x,t) := \rho(x,t)u(x,t) = \sum_i c_i F_i(x,t) \ . \tag{4.7}$$

Again, $u(x,t)$ denotes the particle velocity at position x and time t.

4.2.3 The LBM in 2D

In the 2D case, we exclusively use the common *D2Q9* LBM model. As the name suggests, there are nine distribution functions per lattice site, see Figure 4.1 In the language of CA, which we have introduced in Section 4.1, this D2Q9 LBM model thus employs the Moore neighborhood. For alternative models, we refer to [WG00].

Figure 4.2 shows the update of an individual lattice site for a D2Q9 LBM model. The grid on the left illustrates the source grid corresponding to the previous point in time, the dark arrows and the dot represent the particle distribution function values being read. The grid on the right illustrates the destination grid corresponding to the current point in time, the dark arrows as well as the dot represent the new particle distribution function values; i.e., after the collide operation.

In the D2Q9 case, the discrete lattice velocities c_i are chosen as follows:

$$c_i := \begin{cases} (0,0) \ , & i = \text{C} \ , \\ (0,\pm c) \ , & i = \text{N, S} \ , \\ (\pm c,0) \ , & i = \text{E, W} \ , \\ (c,\pm c) \ , & i = \text{NE, SE} \ , \\ (-c,\pm c) \ , & i = \text{NW, SW} \ . \end{cases}$$

To facilitate further reading, the directions into which these velocities point will be referred to as center (C), north (N), east (E), south (S), west (W), north-east (NE), south-east (SE), south-west (SW), and north-west (NW). We usually choose $c := 1$ by default.

In order that we can compute the new distribution functions $F_i(x, t + \Delta t)$ using (4.5), it is necessary to determine the local equilibrium distribution functions $F_i^{(0)}(x,t)$ beforehand. According to [WG00], the latter are computed as:

$$F_i^{(0)}(x,t) = \frac{4}{9}\rho(x,t)\left(1 - \frac{3}{2}\frac{\langle u(x,t), u(x,t)\rangle}{c^2}\right) \tag{4.8}$$

for $i = \text{C}$,

$$F_i^{(0)}(x,t) = \frac{1}{9}\rho(x,t)\left(1 + 3\frac{\langle c_i, u(x,t)\rangle}{c^2} + \frac{9}{2}\frac{\langle c_i, u(x,t)\rangle^2}{c^4} - \frac{3}{2}\frac{\langle u(x,t), u(x,t)\rangle}{c^2}\right) \tag{4.9}$$

for $i \in \{\text{N, E, S, W}\}$, and

$$F_i^{(0)}(x,t) = \frac{1}{36}\rho(x,t)\left(1 + 3\frac{\langle c_i, u(x,t)\rangle}{c^2} + \frac{9}{2}\frac{\langle c_i, u(x,t)\rangle^2}{c^4} - \frac{3}{2}\frac{\langle u(x,t), u(x,t)\rangle}{c^2}\right) \tag{4.10}$$

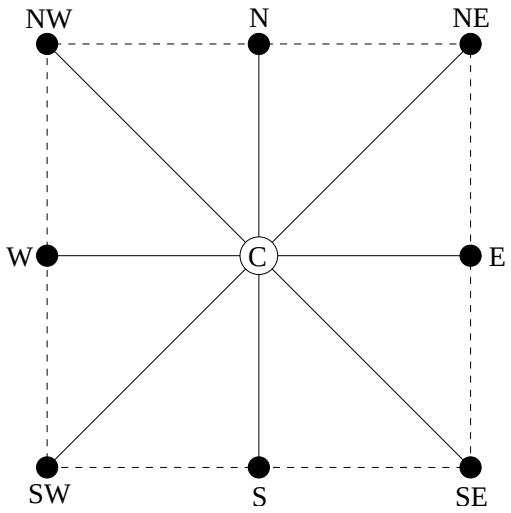


Figure 4.1: D2Q9 lattice model for the LBM.

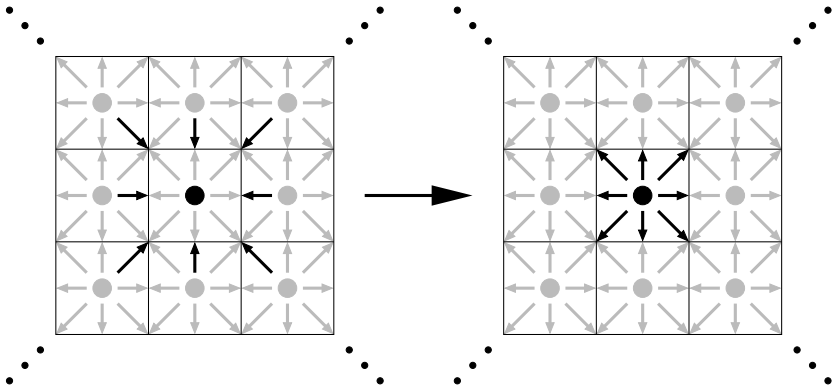


Figure 4.2: Representation of the 2D LBM updating a single lattice site by a stream operation (left) and a subsequent collide operation (right).

for $i \in \{\text{NE, SE, NW, SW}\}$. A detailed discussion of this theory is irrelevant for the understanding of our optimization techniques and therefore beyond the scope of this thesis. Instead, we refer to [HL97, WG00] for a derivation of these equations.

4.2.4 The LBM in 3D

Common cubic lattice models in 3D are *D3Q15*, *D3Q19*, and *D3Q27*. They are characterized by 15, 19, and 27 discrete velocities. Thus, there are 15, 19, and 27 distribution functions per lattice site, respectively. Comparisons of these models can be found in [MSY02, WG00], for example. Figures 4.3, 4.4, and 4.5 illustrate the directions of the discrete velocities in the D3Q15, the D3Q19, and the D3Q27 lattice model, respectively.

According to [MSY02], the D3Q19 model provides a balance between numerical stability (i.e., computational reliability) and efficiency (i.e., computational complexity). We therefore use the D3Q19 model for our implementations of the LBM in 3D and our subsequent performance experiments. In this setting, the lattice velocities c_i are chosen as follows:

$$c_i := \begin{cases} (0,0,0) \ , & i = \text{C} \ , \\ (0,0,\pm c) \ , & i = \text{T, B} \ , \\ (0,\pm c,0) \ , & i = \text{N, S} \ , \\ (\pm c,0,0) \ , & i = \text{E, W} \ , \\ (0,\pm c,\pm c) \ , & i = \text{TN, TS, BN, BS} \ , \\ (\pm c,0,\pm c) \ , & i = \text{TE, TW, BE, BW} \ , \\ (\pm c,\pm c,0) \ , & i = \text{NE, NW, SE, SW} \ . \end{cases}$$

Analogous to the 2D case, in order to facilitate further reading, the directions into which these velocities point will be referred to as center (C), north (N), east (E), south (S), west (W), north-east (NE), south-east (SE), south-west (SW), north-west (NW), top-north (TN), top-south (TS), top-east (TE), top-west (TW), bottom-north (BN), bottom-south (BS), bottom-east (BE), and bottom-west (BW). Again, we choose $c := 1$ by default.

In the case of the D3Q19 model, the equilibrium distributions $F_i^{(0)}(x,t)$ are computed as follows [WG00]:

$$F_i^{(0)}(x,t) = \frac{1}{3}\rho(x,t)\left(1 - \frac{3}{2}\frac{\langle u(x,t), u(x,t)\rangle}{c^2}\right) \tag{4.11}$$

for $i = \text{C}$,

$$F_i^{(0)}(x,t) = \frac{1}{18}\rho(x,t)\left(1 + 3\frac{\langle c_i, u(x,t)\rangle}{c^2} + \frac{9}{2}\frac{\langle c_i, u(x,t)\rangle^2}{c^4} - \frac{3}{2}\frac{\langle u(x,t), u(x,t)\rangle}{c^2}\right) \tag{4.12}$$

for $i \in \{\text{N, E, S, W, T, B}\}$, and

$$F_i^{(0)}(x,t) = \frac{1}{36}\rho(x,t)\left(1 + 3\frac{\langle c_i, u(x,t)\rangle}{c^2} + \frac{9}{2}\frac{\langle c_i, u(x,t)\rangle^2}{c^4} - \frac{3}{2}\frac{\langle u(x,t), u(x,t)\rangle}{c^2}\right) \tag{4.13}$$

for $i \in \{\text{TN, TS, BN, BS, TE, TW, BE, BW, NE, NW, SE, SW}\}$.

4.2.5 Boundary Conditions

Most boundary conditions for a solid boundary used in the LBM are based upon the *bounce-back* assumption (also known as the *no-slip* condition), as opposed to the *free-slip* boundary condition. In the case of a bounce-back boundary, any particles that hit a boundary simply reverse their velocity so that the average velocity at the boundary is automatically zero, as observed experimentally

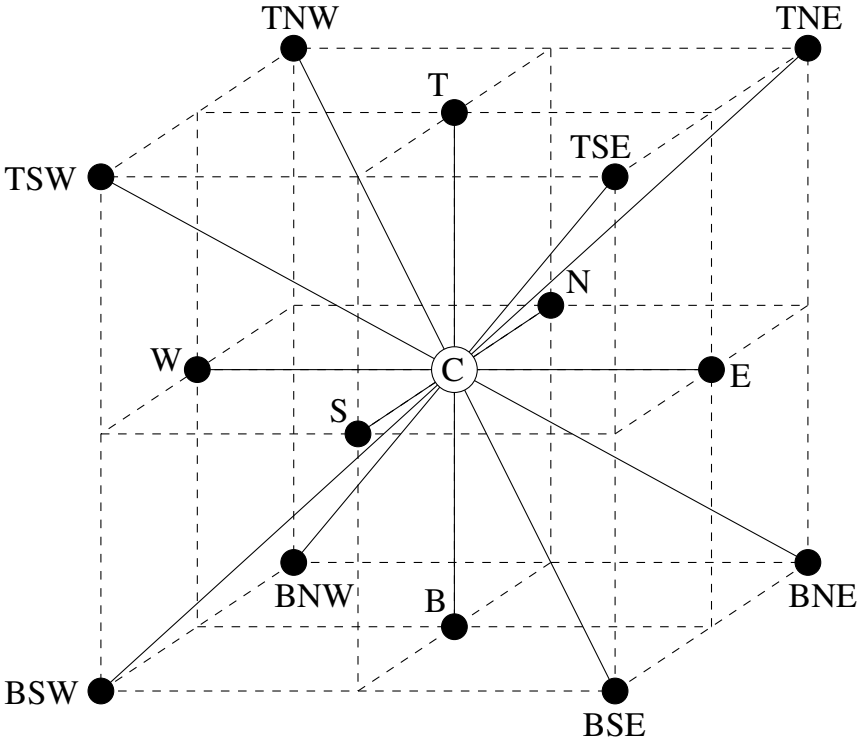


Figure 4.3: D3Q15 lattice model for the LBM.

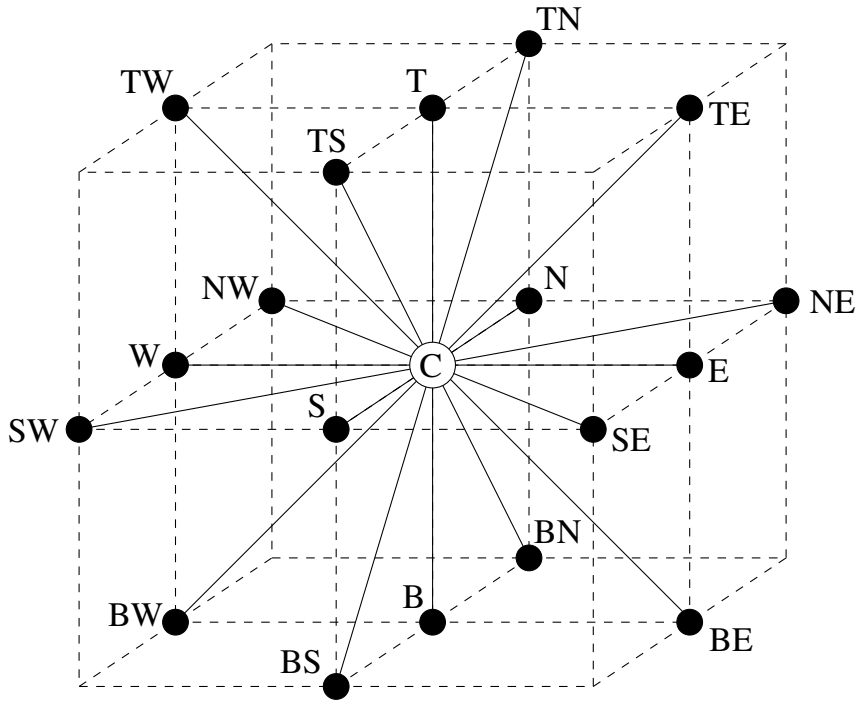


Figure 4.4: D3Q19 lattice model for the LBM.

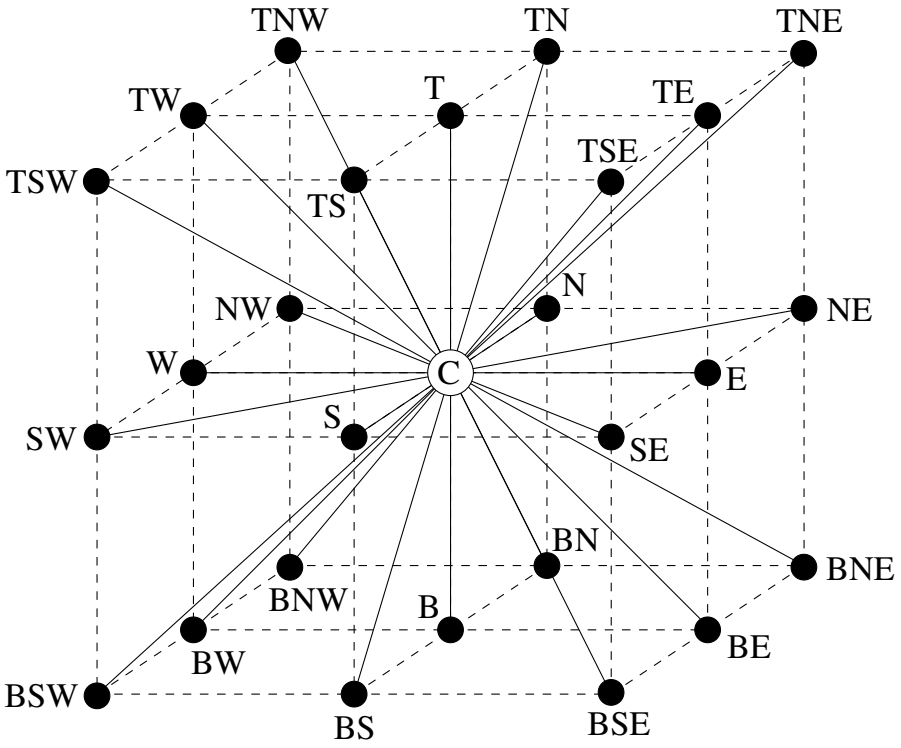


Figure 4.5: D3Q27 lattice model for the LBM.

[CDE94]. Solid obstacles in the interior of the computational domain are usually treated in the same fashion.

This explains why the LBM can easily be used in the case of moving obstacles and complex geometries which may even be time-dependent. Of course, in order to accurately represent complex boundary and obstacle shapes, it is necessary that the grid spacing is chosen fine enough. Unfortunately, this typically means a significant increase in code execution time. A discussion of various approaches concerning the handling of boundary conditions in the LBM can be found in [MSY02, WG00], for example.

4.2.6 Abstract Formulation of the LBM

We now have everything in place to present an abstract formulation of the LBM, see Algorithm 4.1. This algorithmic description is based on general properties of the LBM (see Section 4.2.2) as well as on properties depending on the dimension and the actual lattice model under consideration (see Sections 4.2.3 and 4.2.4). For the sake of simplicity, we have omitted the special handling of boundary cells in Algorithm 4.1.

Note that Algorithm 4.1 starts with a collide operation based on the initial equilibrium distribution functions $F_i^{(0)}(x, t_0)$ *(collide-and-stream update order)*. Alternatively, the LBM could start with a stream step before performing the first collide operation *(stream-and-collide update order)*. However, this would represent a different algorithm and, in general, different numerical results would be computed, at least until some stationary equilibrium would be reached. We will repeatedly return to this subtle difference in the remainder of this thesis.

Algorithm 4.1 Abstract formulation of the LBM, collide-and-stream update order.

1: Using (4.8)–(4.10) in 2D or (4.11)–(4.13) in 3D, calculate the equilibrium distribution functions $F_i^{(0)}(x, t_0)$ based on the initial mass density $\rho(x, t_0)$ and the initial momentum density $j(x, t_0)$
2: $F_i(x, t_0) \leftarrow F_i^{(0)}(x, t_0)$
3: $t \leftarrow t_0$
4: Apply (4.5); i.e., perform a collide operation and subsequent stream operation to determine the new distribution functions $F_i(x, t + \Delta t)$
5: $t \leftarrow t + \Delta t$
6: Use (4.6) and (4.7) to compute the resulting mass density $\rho(x, t)$ and the resulting momentum density $j(x, t)$
7: Compute the equilibrium distribution functions $F_i^{(0)}(x, t)$
8: Go to Step 4 unless the required number of time steps has been performed

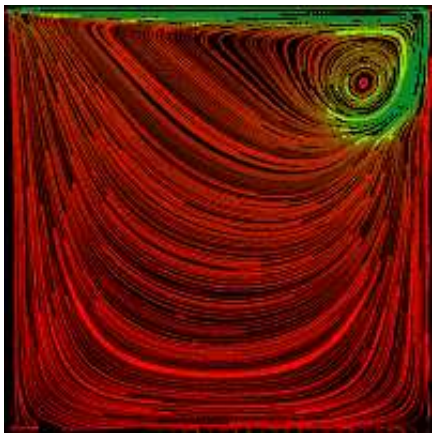
4.2.7 LBM Model Problem: Lid-Driven Cavity

In order to test and benchmark the various implementations of the LBM in 2D and 3D, a well known test case in fluid dynamics known as the *lid-driven cavity* has been used. The system under consideration consists of a closed box, where the top of the box, the lid, is continually dragged in the same direction across the fluid. After a while, the fluid in the box forms a circular flow. Figure 4.6 illustrates the situation in 2D for a grid of 200×200 cells. A more detailed description of this model problem can be found in [GDN98].

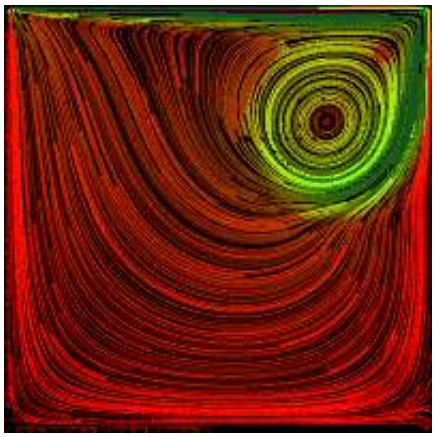
Our implementations of the LBM for the solution of this model problem use three different types of cells; fluid cells, obstacle cells, and acceleration cells. Obstacle cells represent the walls of the box as well as possible solid obstacles in the interior of the box. We have implemented bounce-back boundary conditions in 2D and 3D, cf. Section 4.2.5. This means that the update rule for each obstacle cell simply turns around the incoming particles by 180° instead of performing a collision.

The acceleration cells are introduced to model the lid. They are characterized by a constant mass density $\rho(x, t)$ as well as a constant velocity $u(x, t)$ in the direction in which the lid is moving.

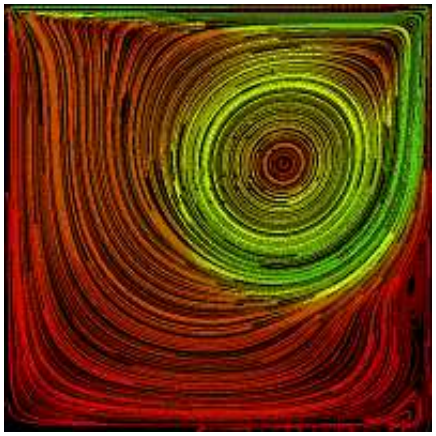
Brief overviews of our LBM codes can be found in Chapter 7, where experimental results will be presented, see particularly Sections 7.4.1 and 7.5.1. For detailed descriptions of our implementations in 2D and 3D, we refer to [Wil03] and to [Igl03], respectively.



(a) After 2000 time steps.

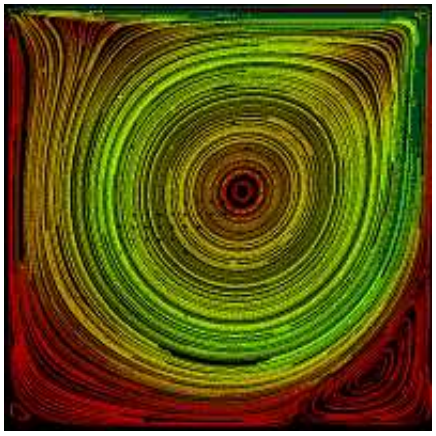


(b) After 4000 time steps.

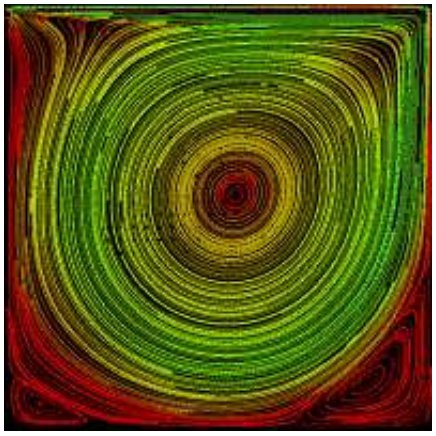


(c) After 8000 time steps.

(d) After 12000 time steps.



(e) After 16000 time steps.



(f) After 20000 time steps.

Figure 4.6: Lid-driven cavity in 2D on a 200^2 lattice.

Part II

Code Optimization Techniques for Memory Hierarchies

Chapter 5

Data Layout Optimizations

5.1 Basics

Data layout optimizations aim at enhancing code performance by improving the arrangement of the data in address space [Tse97]. On one hand, such techniques can be applied to change the mapping of array data to the cache frames, thereby reducing the number of cache conflict misses. This is achieved by a layout transformation called *array padding*, see Section 5.2.

On the other hand, data layout optimizations can be applied to increase spatial locality, cf. Section 2.3. They can be used to reduce the size of the working set of a process; i.e., the number of virtual pages which are referenced alternatingly and should therefore be kept in main memory [HP03]. Furthermore, data layout transformations can be introduced in order to increase the reuse of cache blocks once they have been loaded into cache. As was outlined in Section 2.4.1, cache blocks contain several data items that are arranged next to each other in address space. Hence, it is reasonable to aggregate data items in address space which are likely to be referenced within a short period of time. This approach will be discussed in Section 5.3.

The following presentation of data layout transformations is partly based on our publications [KW02, KW03].

5.2 Array Padding

5.2.1 Motivation and Principle

If two or more arrays are accessed in an alternating manner as in both parts of Algorithm 5.1, and the data items which are needed simultaneously are mapped to the same cache frames, a high number of conflict misses occurs, cf. Section 2.4.4. Note that a Fortran77-like notation has been used for both parts of Algorithm 5.1.

In this example, reading the first element of array a will load a cache line containing this array element and possibly subsequent array elements for further use. Provided that the first array element of array b is mapped to the same cache frame as the first element of array a, a read of the former element will trigger the cache to replace the elements of array a which have just been loaded. The following access to the next element of array a will no longer be satisfied by the cache, thus force the cache to reload the data and in turn to replace the data of array b. Hence, the elements of b must be reloaded, and so on. Although both arrays are referenced sequentially with

Algorithm 5.1 Inter-array padding.

```
1: // Original code:
2: double precision a(1024)
3: double precision b(1024)
4: for i = 1 to 1024 do
5:     sum = sum + a(i) * b(i)
6: end for
```

```
1: // After applying inter-array padding:
2: double precision a(1024)
3: double precision pad(x)
4: double precision b(1024)
5: for i = 1 to 1024 do
6:     sum = sum + a(i) * b(i)
7: end for
```

unit stride[1], no reuse of data which has been preloaded into the cache will occur since the data is evicted immediately by elements of the other array. This phenomenon is called *cross interference* of array references [LRW91]. A similar problem — called *self interference* — can occur if several rows of a multidimensional array are mapped to the same set of cache frames and the rows are accessed in an alternating fashion.

The fundamental problem with multidimensional arrays is that a layout function must be defined which maps the multidimensional index spaces of the arrays to the linearly ordered address space. Typically, this function is linear and maps array elements which are adjacent in some dimension to consecutive locations in address space, whereas array elements which are adjacent in some other dimension are mapped to widely separated locations.

The definition of the layout function depends on the programming language. For example, in Fortran77 the leftmost dimension is the leading dimension (column-major order), whereas in C/C++ the rightmost dimension runs fastest (row-major order). In [DLL01], a semi-hierarchical layout function is presented which is dimension-neutral with respect to the access distance of array elements in memory. In the following, we will constantly assume the standard case of linear array layouts.

For both cases of interference, array padding provides a means to reduce the number of conflict misses [RT98, Riv01]. *Inter-array padding* involves the insertion of unused variables (pads) between two arrays in order to avoid cross interference. Introducing pads modifies the offset of the second array; i.e., the relative distance of the two arrays in address space and therefore the mapping of the array elements to the cache frames.

In contrast, *intra-array padding* inserts unused array elements between rows of such an array by increasing its leading dimension; i.e., the dimension running fastest in memory is increased by a suitable number of extra elements. As we have noted above, the array dimension that runs fastest in address space depends on the layout function defined by the programming language. Obviously, a general disadvantage of array padding techniques is that some extra memory is required for the pads.

The sizes of the pads depend on the mapping scheme of the cache, the cache size, the cache line size, its set associativity, and the data access pattern of the code. Typical padding sizes are multiples of the cache line size, but different sizes may be used as well. Array padding is usually applied at compile time. Intra-array padding can, in principle, be introduced at runtime, if data structures are allocated dynamically. In any case, knowledge of the cache architecture is indispensable, and information about the access pattern of the program will improve the quality of the selected padding size.

There are several approaches to the question of how to choose the size of the pads. Some

[1] The stride is the distance of array elements in address space which are accessed within consecutive loop iterations.

research focuses on the development of suitable heuristics [RT98, RT00, Riv01]. However, this approach is based on simplified models of the memory architecture and often not applicable to realistic simulations where many arrays may be involved simultaneously. As a consequence, the performance results are often not optimal.

A second approach is based on exhaustively searching a reasonable portion of the (potentially high dimensional) parameter space and using those pad sizes yielding the best performance. This approach follows the *AEOS (automated empirical optimization of software)* paradigm [WPD01], which will be presented in more detail in Section 9.1.2. Apparently, the downside to this empirical parameter search is that it is more time-consuming than the application of padding heuristics.

In the following, we will describe the application of intra-array padding for stencil-based iterative methods in 2D and 3D. Since the 2D case has already been presented in detail in [Wei01], we will keep its description short and concentrate on the 3D case instead.

5.2.2 Intra-Array Padding in 2D

Figure 5.1 illustrates the introduction of pads into a 2D array in order to avoid self interference. The leading dimension of the array is increased by a fixed number of unused elements.

Both parts of Algorithm 5.2 illustrate how this is accomplished in Fortran77-like notation, where the first index of the 2D array runs fastest in address space. We refer to [KWR02, Wei01], [Pfä01], and to the references provided therein for details on intra-array and inter-array padding for iterative methods on regular 2D grids.

Algorithm 5.2 Intra-array padding in 2D.

1: *// Original code:*	1: *// After applying intra-array padding:*
2: double precision $a(n_x, n_y)$	2: double precision $a(n_x + p_x, n_y)$
3: $\cdots$	3: $\cdots$

5.2.3 Intra-Array Padding in 3D

5.2.3.1 The Standard Approach

Analogous to the 2D case, intra-array padding can be applied to 3D arrays in order to reduce the number of cache conflict misses resulting from self interference. Typically, intra-array padding is introduced into a 3D array by increasing both its leading dimension as well as its second dimension by appropriately chosen increments p_x and p_y, respectively [Riv01]. This technique is illustrated

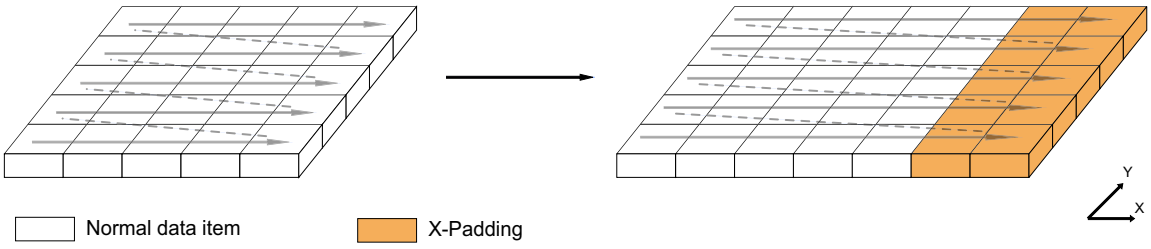


Figure 5.1: Intra-array padding in 2D; left: no padding, right: the leading array dimension has been incremented appropriately.

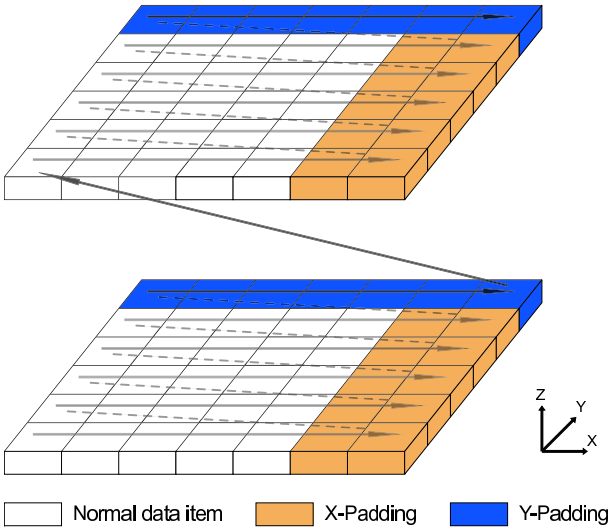


Figure 5.2: Standard intra-array padding in 3D.

in Figure 5.2. The right part of Algorithm 5.3 illustrates how this can be accomplished using a Fortran77-like language, where the leftmost array index runs fastest in address space.

The first increment p_x, which pads the leading array dimension and is denoted as *X-padding* in Figure 5.2, is introduced in order to avoid conflicts caused by references to array elements which are adjacent in direction y; i.e., located in neighboring horizontal rows of a plane. In addition, the second pad p_y, which is denoted as *Y-padding* in Figure 5.2, changes the relative distance of neighboring array planes in address space. It therefore modifies the cache mapping of array elements which are adjacent in direction z.

5.2.3.2 The Nonstandard Approach

We have developed an alternative nonstandard intra-array padding technique for 3D arrays, see also [KRTW02]. Our approach is illustrated in Figure 5.3.

The idea is to explicitly pad the leading array dimension (denoted as *X-padding* in Figure 5.3), while the padding for the second array dimension is introduced implicitly (denoted as *nonstandard Y-padding* in Figure 5.3 or as *inter-plane padding*) by performing appropriate additional array index arithmetic when accessing elements. The relative distance of neighboring array planes is changed by adding or subtracting a fixed offset p_y to the first array index when switching from one array plane to an adjacent one in direction z.

The part on the right of Algorithm 5.4 shows the array declarations for our nonstandard intra-array padding technique. In order to account for the implicitly introduced paddings and to provide enough memory for all array elements, we must allocate an auxiliary array r lying directly behind the actual array a in address space (Step 3). The size of this auxiliary array r must be at least the number of array plane transitions times the number of artificially introduced inter-plane pads;

Algorithm 5.3 Standard intra-array padding in 3D.

1: *// Original code:*	1: *// After applying standard intra-array padding:*
2: double precision $a(n_x, n_y, n_z)$	2: double precision $a(n_x + p_x, n_y + p_y, n_z)$
3: $\cdots$	3: $\cdots$

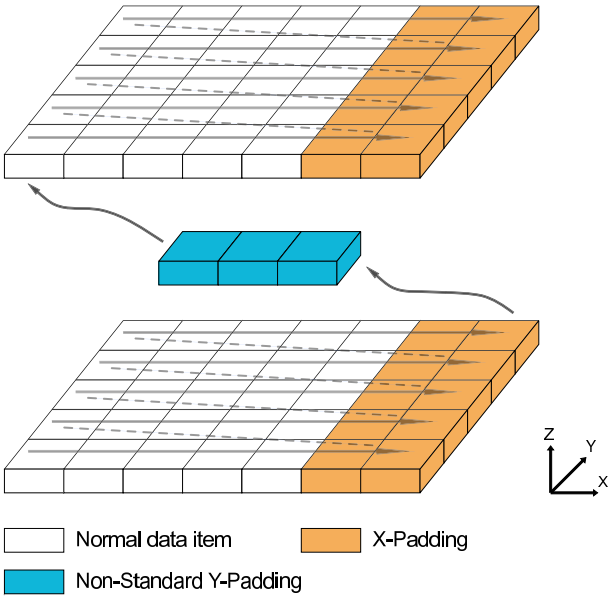


Figure 5.3: Nonstandard intra-array padding in 3D.

i.e., $(n_z - 1)p_y$. The array r is never accessed explicitly, but the manually applied index arithmetic causes r to be accessed via references to the array a.

The artificially introduced displacements between neighboring array planes must be taken into account when accessing neighboring elements of the currently indexed array element at position (x, y, z) which are located in adjacent array planes. For example, the z-adjacent neighbors of the current element at position (x, y, z) must be referenced using the indices $(x + p_y, y, z + 1)$ and $(x - p_y, y, z - 1)$, respectively. As a matter of fact, our nonstandard intra-array padding approach causes some overhead due to the additional index arithmetic[2]. However, if the number of cache conflict misses and therefore the amount of main memory traffic can be reduced significantly, this overhead is almost negligible.

Note that, for this data layout, it is essential that the programming language standard (or at least the underlying compiler) guarantees that the auxiliary array r is allocated next to the array a. Moreover, the language must provide rather flexible array handling. In particular, it is important that no array bound checking is performed in order to enable the inter-plane padding to be introduced, see the examples in Section 5.2.3.3.

The advantage of our nonstandard approach is that the memory overhead is typically less than it is in the case of standard intra-array padding. The number of unused array elements in direction y (i.e., the number of inter-plane pads) is not necessarily a multiple of the sum of the leading array dimension n_x and the corresponding padding p_x, as is the case for standard intra-array padding,

[2]Depending on the capabilities of the optimizing compiler, this overhead may be eliminated through the efficient use of index variables.

Algorithm 5.4 Nonstandard intra-array padding in 3D.

1: *// Original code:*	1: *// After applying nonstandard intra-array padding:*
2: double precision $a(n_x, n_y, n_z)$	2: double precision $a(n_x + p_x, n_y, n_z)$
3: $\cdots$	3: double precision $r((n_z - 1) * p_y)$
	4: $\cdots$

see Figure 5.2.

All padding configurations which can be achieved using the standard intra-array padding approach for 3D arrays can also be achieved using our nonstandard technique. In order to represent any array padding configuration resulting from the standard approach, we must simply choose the inter-plane padding p_y in the nonstandard case to be the corresponding multiple of the (padded) leading array dimension $n_x + p_x$. This follows from a comparison of the Figures 5.2 and 5.3.

5.2.3.3 Example: Nonstandard Intra-Array Padding for Gauss-Seidel

Algorithm 5.5 shows the major part of a conventional implementation of the method of Gauss-Seidel for the solution of Laplace's equation $\Delta u = 0$ on a cubic domain, cf. both Sections 3.2.3 and 3.3.2. We assume a second-order finite difference discretization of the Laplacian on an equidistant 3D grid. This approach yields a 7-point stencil with constant coefficients for each interior node. We further assume Dirichlet boundary conditions. We refer to the brief overview in Section 3.1 and the corresponding references.

Note that we assume Fortran77-like array semantics in these examples; i.e., 1-based arrays. Therefore the loops along the spatial dimensions, which traverse the interior nodes of the grid, run from 2 to $n - 1$ (Steps 6 to 8).

The introduction of standard intra-array padding is trivial, cf. Section 5.2.3.1. We focus on our nonstandard intra-array padding approach instead. Algorithm 5.6 demonstrates the application of this technique to the standard implementation of this computational kernel presented in Algorithm 5.5. As was explained in Section 5.2.3.2, the inter-plane pads are accounted for by manually adapting the first array index (Step 10) and by allocating a second array r (Step 2) lying behind the actual array u in address space.

Alternatively, the range of the innermost loop can be adapted to the current array plane. This version of the code is presented in Algorithm 5.7. Step 9 illustrates how the range of the innermost loop is adapted appropriately. Note that the transformation from Algorithm 5.6 to Algorithm 5.7 reduces the number of integer operations in the body of the innermost loop; compare Step 10 of Algorithm 5.6 with Step 10 of Algorithm 5.7.

Algorithm 5.5 Gauss-Seidel kernel without array padding in 3D.

```
1:  double precision u(n, n, n)
2:  // Initialization of u, particularly the values on the grid boundaries:
3:  ···
4:  // Iterative solution process:
5:  for i = 1 to i_max do
6:    for z = 2 to n − 1 do
7:      for y = 2 to n − 1 do
8:        for x = 2 to n − 1 do
9:          u(x, y, z) = 1/6 ∗ (u(x, y, z − 1) + u(x, y, z + 1) + u(x, y − 1, z) + u(x, y + 1, z)+
            u(x − 1, y, z) + u(x + 1, y, z))
10:       end for
11:     end for
12:   end for
13: end for
```

Algorithm 5.6 Gauss-Seidel kernel with nonstandard intra-array padding in 3D, Version 1.

```
1: double precision u(n + p_x, n, n)
2: double precision r((n − 1) ∗ p_y)
3: // Initialization of u, particularly the values on the grid boundaries:
4: ···
5: // Iterative solution process:
6: for i = 1 to i_max do
7:   for z = 2 to n − 1 do
8:     for y = 2 to n − 1 do
9:       for x = 2 to n − 1 do
10:        u(x+(z−1)∗p_y, y, z) = 1/6∗(u(x+(z−2)∗p_y, y, z−1)+u(x+z∗p_y, y, z+1)+u(x+(z−1)∗
           p_y, y−1, z)+u(x+(z−1)∗p_y, y+1, z)+u(x−1+(z−1)∗p_y, y, z)+u(x+1+(z−1)∗p_y, y, z))
11:      end for
12:    end for
13:  end for
14: end for
```

Algorithm 5.7 Gauss-Seidel kernel with nonstandard intra-array padding in 3D, Version 2.

```
1: double precision u(n + p_x, n, n)
2: double precision r((n − 1) ∗ p_y)
3: // Initialization of u, particularly the values on the grid boundaries:
4: ···
5: // Iterative solution process:
6: for i = 1 to i_max do
7:   for z = 2 to n − 1 do
8:     for y = 2 to n − 1 do
9:       for x = 2 + (z − 1) ∗ p_y to n − 1 + (z − 1) ∗ p_y do
10:        u(x, y, z) = 1/6 ∗ (u(x − p_y, y, z − 1) + u(x + p_y, y, z + 1) + u(x, y − 1, z) + u(x, y + 1, z) +
           u(x − 1, y, z) + u(x + 1, y, z))
11:      end for
12:    end for
13:  end for
14: end for
```

5.3 Cache-Optimized Data Layouts

5.3.1 Array Merging

In the following, we will discuss cache-efficient data layouts for both iterative linear solvers on regular data structures as well as lattice Boltzmann automata. These data layouts are primarily based on *array merging (group-and-transpose)* [GH01, Tse97].

The idea behind array merging is to aggregate data items in memory which are accessed within a short period of time. As a consequence, this technique improves the spatial locality exhibited by elements of different arrays or other data structures and therefore the reuse of cache contents[3]. Additionally, array merging can reduce the number of conflict misses for scenarios with large arrays and alternating access patterns, cf. Section 5.2.

Both parts of Algorithm 5.8 illustrate the application of the array merging technique, again using Fortran77-like notation. For the data layout on the left, the distance of any two array elements $a(i)$ and $b(i)$ in address space is 1000 times the size of a double precision floating-point value. However, for the data layout on the right, these data items become adjacent, if $a(i)$ and $b(i)$ are mapped to $ab(1, i)$ and $ab(2, i)$, respectively.

Note that the reverse of the array merging technique may enhance spatial locality as well. This is the case if compound data structures, which are never accessed as a whole, are split into smaller components in order to save space in cache. This technique is called *split-and-scatter* [Tse97]. Furthermore, accessing separate data structures with a simpler access pattern might leverage hardware prefetching mechanisms on certain CPU architectures. This is another essential aspect that must be taken into account.

Algorithm 5.8 Array merging.

1: *// Original data structure:*	1: *// Merged data structure:*
2: double precision $a(1000)$	2: double precision $ab(2, 1000)$
3: double precision $b(1000)$	

5.3.2 Cache-Optimized Data Structures for Iterative Linear Solvers

Figure 5.4 illustrates three different data layouts for iterative solvers in 2D. These data layouts can be employed to implement Gauss-Seidel-type schemes such as the method of Gauss-Seidel itself and SOR. These schemes have in common that they permit the components of the solution vector to be updated in place, cf. Section 3.2. Therefore, it is sufficient to provide memory for a single copy of the corresponding vector $u^{(k)}$ of unknowns.

The data structures presented in Figure 5.4 are based on the assumption of a discrete operator involving 5-point stencils. Therefore, we need to store the vector of unknowns (U_i), five entries per matrix row ($N_i, E_i, S_i.W_i, C_i$), as well as the right-hand side (F_i). Note that alternative discretizations based on 9-point stencils, for example, can be handled analogously.

Three different data storage schemes are introduced here, see also [KWR02] and [Pfä01]:

- In the case of the commonly used *band-wise* data layout, the five individual bands of the matrix, the vector of unknowns, and the right-hand side are stored separately.

[3] Array merging can also be employed to optimize for larger levels of the memory hierarchy. For instance, this technique can reduce the size of the *active set* of virtual pages; i.e., the number of virtual pages which need to be kept simultaneously in physical main memory.

- The *access-oriented* layout is based on an observation which is true for all iterative schemes we have presented in Section 3.2: whenever an approximation u_i is updated, we need to access the approximations u_j at its neighboring grid nodes as well as all entries $a_{i,j}$ of the corresponding matrix row and the respective component f_i of the right-hand side. Therefore, it is obvious to *merge* the entries $a_{i,j}$ and the right-hand side f_i into a compound data structure. This leads to the access-oriented layout.

- Finally, the *equation-oriented* layout is based on the idea to merge the data which belongs to a single equation in the linear system into a compound data structure. Consequently, for the i-th equation, this data structure contains all entries $a_{i,j}$ of the i-th matrix row, the corresponding right-hand side f_i, and the approximation u_i itself.

Note that similar data layouts can be used for Jacobi-type schemes such as Jacobi's method itself and JOR. However, the implementation of Jacobi-type algorithms parallels the implementation of CA, see Section 5.3.3: approximations (in the case of linear solvers) and state variables (in the case of CA) which will still be needed for future computations must not be overwritten as soon as updates become available. Therefore, additional memory space must be provided in order that these data dependences can be preserved. An explicit discussion of data structures for Jacobi-type methods is omitted here.

The aforementioned data layouts can easily be extended to the 3D case. Figure 5.5 illustrates the corresponding 3D counterparts for Gauss-Seidel-type iterative schemes. The data structures presented in Figure 5.5 assume a discrete operator based on 7-point stencils. Thus, we need to store the vector of unknowns (U_i), seven entries per matrix row ($N_i, E_i, S_i, W_i, T_i, B_i, C_i$), and the right-hand side (F_i). Likewise, alternative discrete operators involving 27-point stencils, for example, can be handled analogously.

A discussion of performance experiments concerning different data layouts for the 2D case can be found in [Pfä01] and in more compact form in [KWR02]. Results for stencil-based iterative

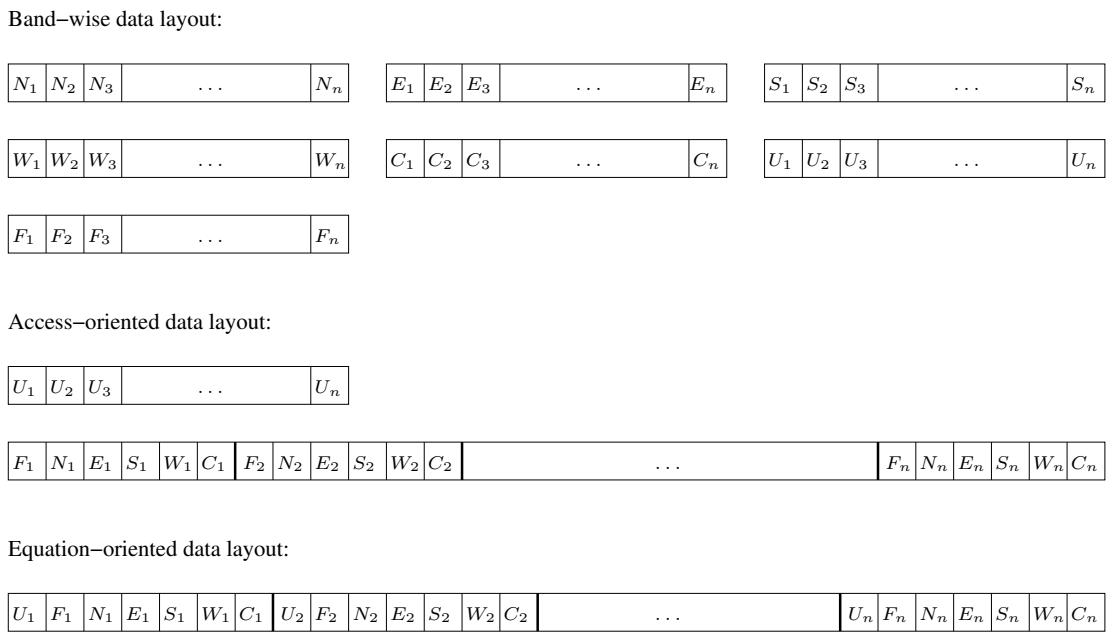


Figure 5.4: Alternative data layouts for Gauss-Seidel-type iterative solvers in 2D.

solvers on 3D grid structures have been presented in [KRTW02] and can also be found in [Thü02]. Furthermore, experimental results concerning the implementation of cache-aware grid structures for the 3D case will be presented in Section 7.3.2.2.

5.3.3 Cache-Optimized Data Structures for CA

We use the LBM as a relevant practical example of CA. However, the data layout optimizations we present in this section are rather general and therefore suitable for all similar CA models.

In the LBM case, the array merging technique can be applied twice. First, the distribution functions that characterize the states of the individual grid cells can be aggregated to records. This corresponds to the access-oriented data storage scheme for iterative linear solvers presented in the previous section. The size of the resulting cell data structure depends on the underlying LBM model. For example, 19 floating-point values representing the distribution functions are aggregated in the case of a D3Q19 lattice, see Section 4.2. Furthermore, we have added a flag (which is represented as another floating-point value) that indicates the type of the cell; e.g., obstacle (solid) cell or fluid cell.

Unfortunately, there is a downside to this data layout. It implies that the streaming step involves noncontiguous memory accesses to neighboring cells. In the case of the stream-and-collide update order, the distribution functions need to be read from the neighboring lattice sites. In contrast, if the collide-and-stream update order is used, the new distribution functions must be written to the neighboring cells. In either case, the memory addresses to be accessed are widely spread in memory. Recall Section 4.2.6 for a comparison of the two alternative update orders for the LBM.

An improved data layout for the LBM has been presented in [DHBW04]. This approach is similar to the band-wise data layout for stencil-based matrices from Section 5.3.2. In addition, it introduces auxiliary arrays in order to store intermediate results. The use of these auxiliary arrays causes the streaming step to perform local memory accesses only, thus increasing spatial locality.

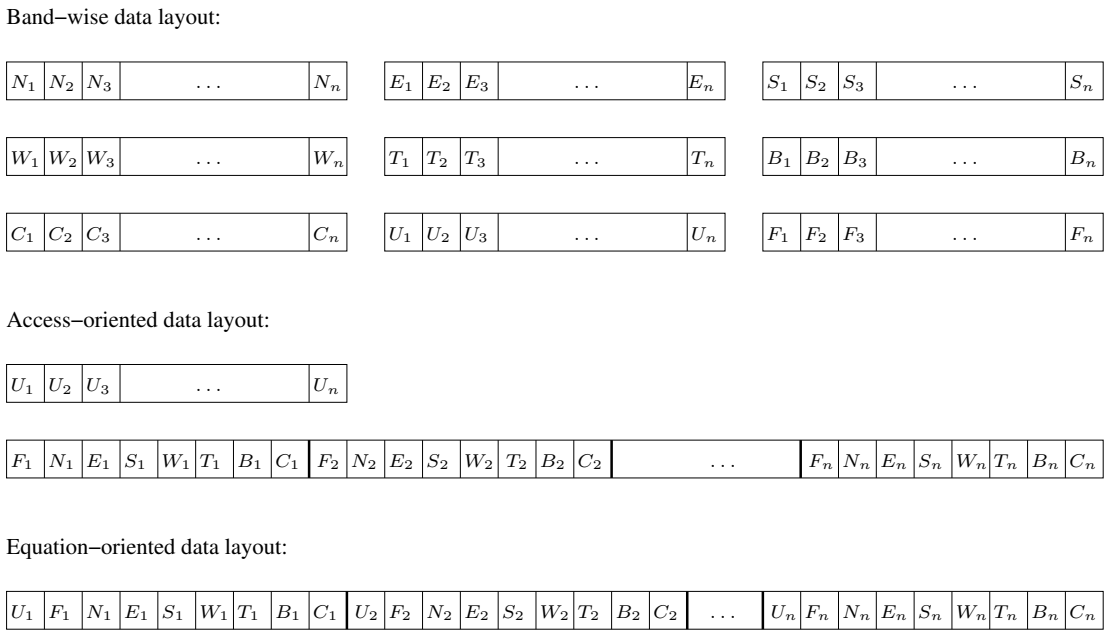


Figure 5.5: Alternative data layouts for Gauss-Seidel-type iterative solvers in 3D.

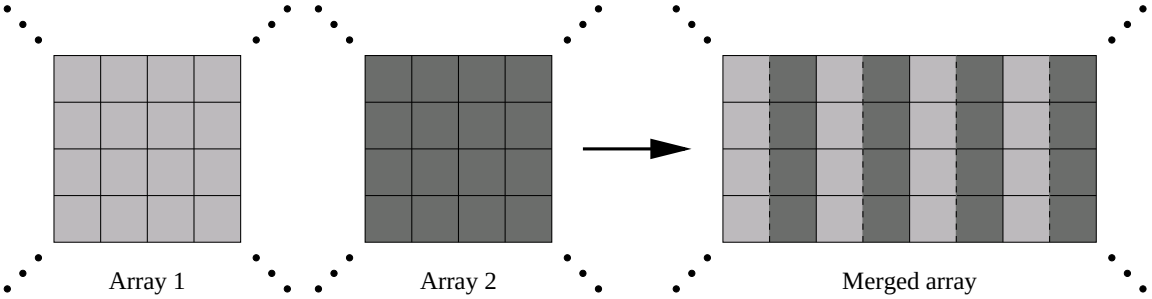


Figure 5.6: Array merging for the LBM in 2D.

As a second application of the array merging technique, the source grid and the destination grid can be interleaved. We use the term *grid merging* to denote this step. Again, the goal is to increase the spatial locality of the code.

Figure 5.6 illustrates how the two arrays which store the grid data at time t and time $t + 1$, respectively, can be merged into a single array. The resulting data structure thus represents an array of pairs of cells where the components of each pair characterize the states of the corresponding lattice site at time t and time $t + 1$, respectively. The extension of this second array merging step to 3D is straightforward.

5.4 Grid Compression for CA and Jacobi-type Algorithms

Grid compression represents another data layout optimization technique that can be applied to both CA as well as to Jacobi-type iterative schemes for solving linear systems that result from stencil-based discretizations of PDEs. These algorithms are characterized by a local update pattern, where each cell or unknown is updated using the values of the cells or unknowns in its neighborhood at the previous time step or iteration. Recall (3.12) from Section 3.2.2 and (4.1) from Section 4.1 for the cases of Jacobi's method and CA, respectively.

The objective of this data layout optimization is to save memory and to increase spatial locality. The fundamental idea is based on the observation that it is not necessary to store the full grid of cells (in the CA case) or the full unknown vector (in the case of Jacobi-type iterative solvers) twice. Rather, it is enough to provide additional buffer space such that data that will still be needed in the future is not immediately overwritten as soon as updates become available.

For ease of presentation, we concentrate on the 2D LBM in the description below, see also [Wil03] and [WPKR03b, WPKR03a]. Like in the grid merging case, however, the extension of this technique to the LBM in 3D is straightforward, see [Igl03] and [PKW$^+$03a, PKW$^+$03b]. Furthermore, it is obvious how the grid compression optimization can be applied to Jacobi-type iterative solvers with local access patterns. A comparable optimization approach for stencil-based iterative schemes has been presented in [BDQ98]. It combines a similar memory-saving technique, which does not require the entire vector of unknowns to be allocated twice, with loop blocking, see Section 6.4.

In an implementation of the 2D LBM, nearly half of the memory can be saved by exploiting the fact that only some data from the eight neighboring cells and the current cell itself is required in order to calculate the new distribution function values at any regular grid site[4]. It is therefore

[4]For the sake of simplicity, we focus on regular (interior) sites and omit the description of how to treat boundary sites and obstacle sites separately.

possible to overlay the two grids for any two consecutive points in time, introducing a diagonal shift of one row and one column of cells into the stream-and-collide or the collide-and-stream operation. The direction of the shift determines the update sequence in each time step since, as we have mentioned previously, we may not yet overwrite those distribution function values that are still required in subsequent lattice site updates.

In Figure 5.7, the light gray area contains the values for the current time t. The two figures illustrate the *fused* stream-and-collide operation[5] for a single cell: the values for time $t + 1$ will be stored with a diagonal shift to the lower left. The fused stream-and-collide sweep must start with the cell located in the lower left corner of the grid in order to preserve data dependences. Consequently, after one complete sweep, the new values (time $t + 1$) are shifted compared to the previous ones (time t). For the subsequent time step, however, the sweep must start with the cell in the upper right corner. The data for time $t + 2$ must then be stored with a shift to the upper right. After two successive sweeps (i.e., after the computation of two successive time steps), the memory locations of the distribution functions are the same as before.

Alternatively, the LBM may start with a collide step instead of a stream step, see Section 4.2.6. Analogously, our grid compression technique can be applied in the case of fused collide-and-stream sweeps as well. As is the case with fused stream-and-collide sweeps, alternating diagonal shifts must be introduced before distribution functions are copied into neighboring lattice sites. These shifts are necessary to avoid overwriting distribution functions that are still needed for future cell updates. Again, after the execution of two successive time steps, the memory locations of the distribution functions are the same as before.

The alternating scheme caused by the grid compression approach is illustrated in Figure 5.8. Here, an 11×11 lattice is used. The grid compression technique thus requires only $12 \times 12 = 144$ cells instead of $2 \times 11 \times 11 = 242$ cells to be allocated.

More generally, in the case of a CA in 2D with $n_x \times n_y$ grid cells, the grid compression technique requires only $(n_x + 1)(n_y + 1)$ cells instead of $2n_xn_y$ cells to be allocated. Correspondingly, if the grid compression approach is used for CA with $n_x \times n_y \times n_z$ cells in 3D, only $(n_x+1)(n_y+1)(n_z+1)$ instead of $2n_xn_yn_z$ cells need to be stored in memory. However, additional buffer space must be provided if several time steps (in the case of CA) or Jacobi-type iterations (in the case of linear solvers) are blocked into a single pass through the grid in order to enhance temporal locality and therefore the reuse of cache contents. We will return to this issue in Section 6.4 in the context of loop blocking for the LBM.

Generally speaking, in the case of an $n \times n$ CA in 2D, our grid compression technique requires the allocation of *additional* buffer space of only $\mathcal{O}(n)$ cells, while the other layout schemes, which are based on the allocation of two full grids, obviously require $\mathcal{O}(n^2)$ *additional* cells to be allocated. Likewise, for an $n \times n \times n$ CA in 3D, grid compression implies an *additional* memory overhead of only $\mathcal{O}(n^2)$ cells, while storing two full grids corresponds to an *additional* number of $\mathcal{O}(n^3)$ grid cells which must be allocated as buffer space.

[5] A description of the *loop fusion* technique for the LBM case will be given in Section 6.3.3.

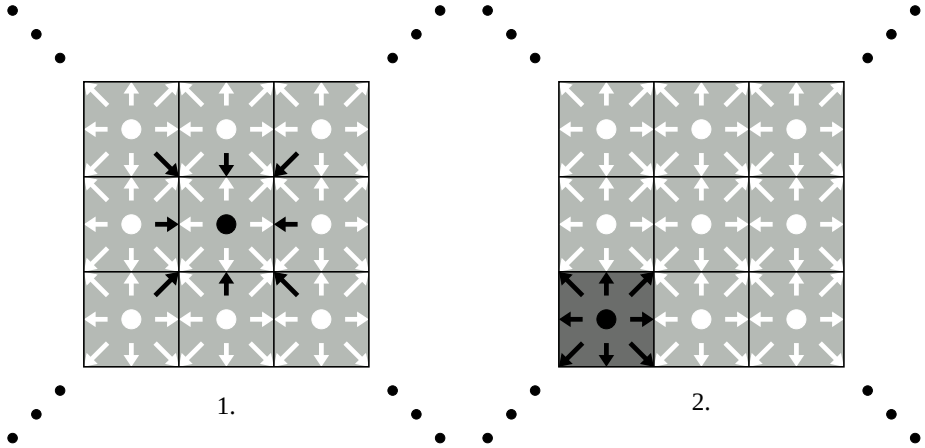


Figure 5.7: Fused stream-and-collide step for the cell in the middle in the context of grid compression for the LBM in 2D; note the diagonal shift to the lower left.

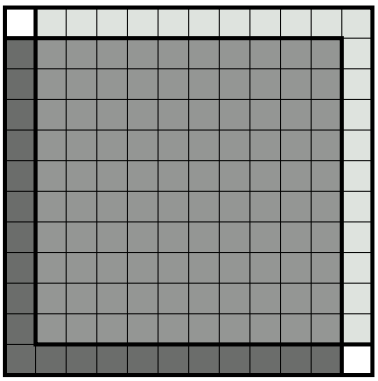


Figure 5.8: Layout of two overlayed 11×11 grids after applying the grid compression technique to the LBM in 2D.

Chapter 6

Data Access Optimizations

6.1 Basics

Data access optimizations are code transformations which change the order in which iterations in a loop nest are executed. These transformations primarily strive to improve both spatial and temporal locality. Moreover, they can also expose parallelism and make loop iterations vectorizable. Note that the data access optimizations we present in this chapter maintain all data dependences and do not change the results of the numerical computations[1].

Usually, it is difficult to decide which combination of code transformations must be applied in order to achieve the maximal performance gain. Compilers typically use heuristics or cost models in order to determine whether a transformation will be effective or not. See [CHCM03], for example.

Loop transformation theory and algorithms found in the literature commonly focus on transformations for *perfectly nested loops* [AMP00]; i.e., nested loops where all assignment statements are contained in the innermost loop. However, loop nests in scientific codes are not perfectly nested in general. Hence, initial enabling transformations are required; e.g., loop skewing and loop peeling. Descriptions of these transformations can be found in the standard compiler literature. See [AK01, BGS94, Muc97, Wol96], for example. In particular, a formal framework for the derivation and evaluation of transformations for loop nests is presented in [WL91]. A combined transformation approach that targets both the automatic parallelization of regular loop nests for shared-memory architectures (e.g., symmetric multiprocessors) and the simultaneous optimization of data cache performance is presented in [Slo99].

A set of general data locality transformations to enhance the temporal locality of codes is described in [PMHC03]. Unlike the approaches that we focus on, these transformations are not restricted to loop-based numerical computations. They cover cache optimization techniques based on early execution as well as deferred execution. We particularly refer to the list of further references included in [PMHC03].

In the following sections, which are partly based on our publications [KW02, KW03], a set of loop transformations will be described which focus on improving data locality for one level of the memory hierarchy; typically a cache level. As was mentioned in Section 1.2.1, instruction cache misses do not have a severe impact on the performance of numerically intensive codes since these programs typically execute small computational kernels repeatedly.

[1] However, these transformations may trigger an aggressively optimizing compiler to reorder floating-point operations. Due to the properties of finite precision arithmetic, this may cause different numerical results.

For ease of presentation, the illustrations in this chapter show relatively small grids only. Furthermore, in the descriptions of our loop blocking approaches, artificially small block sizes are used in order to keep the figures as clear and simple as possible. In a realistic application, of course, the block sizes must be adapted to the properties of the cache level (e.g., its capacity) for which the code is tailored.

6.2 Loop Interchange

6.2.1 Motivation and Principle

This transformation reverses the order of two adjacent loops in a loop nest [AK01, Wol96]. Generally speaking, loop interchange can be applied if the order of the loop execution is unimportant. Loop interchange can be generalized to *loop permutation* by allowing more than two loops to be moved at once and by not requiring them to be adjacent.

Loop interchange can improve spatial locality by reducing the stride of an array-based computation[2]. Upon a memory reference, several words of an array are loaded into a cache line, cf. Section 2.4.1. If the array is larger than the cache, accesses with large stride will only use one word per cache line. The other words which are loaded into the cache frame are evicted before they can be reused.

Loop interchange can also be used to enable vectorization, to expose parallelism, and to improve register reuse. However, the different targets may be conflicting. For example, increasing parallelism requires loops with no dependences to be moved outward, whereas vectorization requires them to be moved inward.

The effect of loop interchange is illustrated in Figure 6.1. We assume that the $(6, 10)$ array is stored in memory in column-major order; i.e., two array elements are stored next to each other in memory if their first indices are consecutive numbers, as is the case for Fortran77, for example (cf. Section 5.2.1). We further assume that each cache line holds four array elements. The code corresponding to the left part of Figure 6.1 accesses the array elements in a row-wise manner, thus using a stride of six elements. Consequently, the preloaded data in the cache line (marked with gray color) will not be reused if the array is too large to fit entirely in cache. However, after interchanging the loop nest, the array is no longer accessed using a stride of six array elements, but unit stride.

[2]Hence, loop interchange often corresponds to a particular data layout optimization which is called *array transpose* [Wei01].

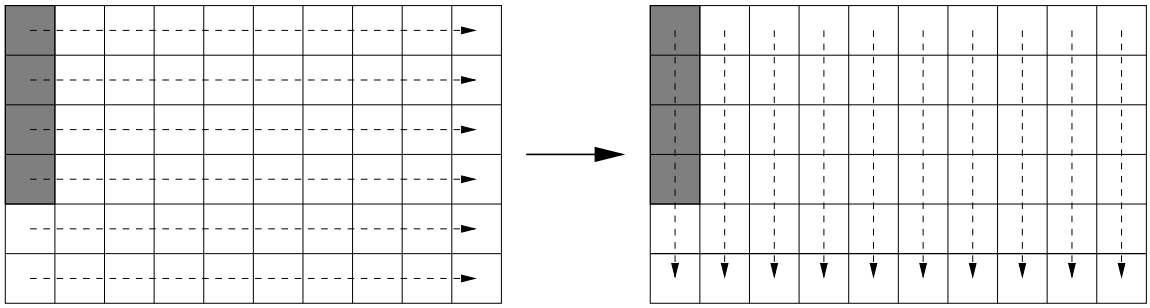


Figure 6.1: Access patterns for interchanged loop nests.

Hence, all words in the cache line are now used by successive loop iterations. This access pattern is illustrated in the right part of Figure 6.1.

As an example, the application of loop interchange is demonstrated by the transition from the left part to the right part in Algorithm 6.1. Again, we assume column-major order.

6.2.2 Loop Interchange for Iterative Linear Solvers

In this section, we refer to the basic iterative schemes which we have introduced in Section 3.2. The actual iterative method under consideration determines whether the order of the loops may be interchanged or not without changing the numerical results of the computations.

In the case of the methods of Jacobi and JOR, the order in which the unknowns are updated is irrelevant, since the computation of any approximation $u_i^{(k+1)}$ only requires approximations $u_j^{(k)}$; i.e., approximations from the previous iteration, cf. (3.12) and (3.14) in Sections 3.2.2 and 3.2.4, respectively. Hence, the numerical results do not depend on the ordering of the unknowns which in turn means that the best possible loop order — in terms of cache utilization — may be chosen.

In the case of the methods of Gauss-Seidel and SOR, however, the order of the loops (i.e., the ordering of the unknowns) is essential. This is due to the fact that the computation of any new approximation $u_i^{(k+1)}$ requires approximations $u_j^{(k)}$ from the previous iteration as well as approximations $u_j^{(k+1)}$ from the current one. See (3.13) and (3.15) in Sections 3.2.3 and 3.2.4, respectively. Therefore, implementing a different loop order means introducing different data dependences and, in general, changing the numerical results.

From the viewpoint of numerical linear algebra, changing the ordering of the unknowns and thereby introducing different data dependences may be a legal and reasonable optimization, even if different numerical results are obtained in the course of the computation, see Section 3.2. However, such optimizing transformations typically require careful mathematical analysis and are a long way from being automatically introduced by an optimizing compiler.

Performance results for loop interchange transformations for red-black Gauss-Seidel in 2D have already been presented in [Pfä01] and in [Wei01]. Experimental results for the 3D case will be presented in Section 7.3.2.3.

6.2.3 Loop Interchange for CA and Jacobi-type Methods

In the case of CA, the loop order does not matter from a theoretical point of view. This can easily be derived from the fact that the new state S_i^{t+1} of any grid cell i at time $t+1$ is computed from its own state S_i^t and the states S_j^t of its surrounding cells (i.e., of its neighborhood) at time t, cf. Section 4.1. Hence, the loop order exhibiting the best performance can be implemented without

Algorithm 6.1 Loop interchange.

```
1: double precision sum
2: double precision a(n,n)
3: // Original loop nest:
4: for i = 1 to n do
5:    for j = 1 to n do
6:       sum = sum + a(i,j)
7:    end for
8: end for
```

```
1: double precision sum
2: double precision a(n,n)
3: // Interchanged loop nest:
4: for j = 1 to n do
5:    for i = 1 to n do
6:       sum = sum + a(i,j)
7:    end for
8: end for
```

influencing the results of the simulation.

As we have already mentioned, the data access pattern of a CA thus corresponds to the data access pattern of a Jacobi-type iteration method and the corresponding remarks from the previous section carry over to the case of CA.

6.3 Loop Fusion

6.3.1 Motivation and Principle

Loop fusion (loop jamming) is a transformation which takes two adjacent loops that have the same iteration space traversal and combines their bodies into a single loop. Loop fusion is the inverse transformation of *loop distribution (loop fission)* which breaks a single loop into multiple loops with the same iteration space. Generally speaking, loop fusion is legal as long as no data dependences between the original loops are violated. See [AK01] for formal details on loop transformation theory.

Fusing two loops results in a single loop which contains more instructions in its body and therefore offers increased instruction level parallelism. Furthermore, only one loop is executed, thus reducing the total loop overhead by approximately a factor of 2.

Most important of all, loop fusion can improve data locality. Assume that two consecutive loops perform global sweeps through an array (as is the case for the code shown in the left part of Algorithm 6.2) and that the data of the array is too large to fit completely in cache. The data of array b which is loaded into the cache by the first loop will not completely remain in cache, and the second loop will have to reload the same data from main memory. If, however, the two loops are combined with loop fusion, only one global sweep through the array b will be performed, see the right part of Algorithm 6.2. Consequently, fewer cache misses will occur.

6.3.2 Application of Loop Fusion to Red-Black Gauss-Seidel

In this section, we consider a standard implementation of red-black Gauss-Seidel, cf. Algorithm 3.1 in Section 3.2.3. We assume that the linear system to be solved results from the discretization of an underlying PDE, which is either based on 5-point stencils in 2D or on 7-point stencils in 3D. As was mentioned in Section 3.2.3, the values at the red nodes only depend on the values at their black neighbor nodes, and vice versa.

In this situation, the loop fusion technique can be applied in order to fuse the sweep over all red nodes and the subsequent sweep over all black nodes into a single sweep over all nodes, replacing two passes over the grid by a single one[3]. The idea behind this optimization step is based on the observation that any black node may be updated as soon as all of its red neighbors have been updated. This data dependence does not require all red nodes to be updated before the beginning of the update sweep over the black nodes.

The application of the loop fusion technique to red-black Gauss-Seidel in 2D has already been presented in [Wei01, WKKR99] and, for the case of variable coefficients, in [Pfä01] and in [KWR02]. Hence, we restrict our discussion to the 3D case involving 7-point stencils, see also [KRTW02] and [Thü02]. Matrices with such band structures arise from second-order finite difference discretizations

[3]Before the actual loop fusion technique can be applied, a shift operation must be introduced such that the second loop nest, which traverses all black nodes, has the same iteration space as the first loop nest, which traverses all red nodes, see Algorithm 6.3.

Algorithm 6.2 Loop fusion.

```
// Original code:
for i = 1 to n do
    b(i) = a(i) + 1.0
end for
for i = 1 to n do
    c(i) = b(i) * 4.0
end for
```

```
// After loop fusion:
for i = 1 to n do
    b(i) = a(i) + 1.0
    c(i) = b(i) * 4.0
end for
```

of the Laplacian, for example. Since the application of loop fusion to red-black Gauss-Seidel in 3D is a natural extension of its 2D counterpart, we will keep the description short.

A standard implementation of red-black Gauss-Seidel in 3D is shown in Algorithm 6.3. Note that this formulation is derived from the more abstract representation provided in Algorithm 3.1. We assume that the 3D grid function is stored as a 3D array which is 0-based. Furthermore, we assume Dirichlet boundary values which are stored explicitly such that the update pattern of each interior node is characterized by a 7-point stencil and no special handling is needed for those interior nodes that are located close to the boundary. Interior nodes have indices (x, y, z), where $0 < x < n_x$, $0 < y < n_y$, and $0 < z < n_z$.

Typically, the grid is too large to fit completely in any of the cache levels of the underlying memory hierarchy. Therefore, much of the data needed during the update sweep over the black nodes (i.e., current approximations at the neighboring red nodes) has already been evicted from cache before it is actually reused. Hence, the standard implementation of red-black Gauss-Seidel requires the unknown vector to be passed through the memory hierarchy twice per iteration.

The application of the loop fusion technique to red-black Gauss-Seidel in 3D is illustrated in Figure 6.2, which shows two adjacent planes (denoted as *Plane* i and *Plane* $i - 1$) in a 3D grid.

Algorithm 6.3 Standard red-black Gauss-Seidel in 3D.

```
for i = 1 to i_max do
  // Update red nodes:
  for z = 1 to n_z − 1 do
    for y = 1 to n_y − 1 do
      for x = 2 − mod (z + y, 2) to n_x − 1 by 2 do
        update(x, y, z)
      end for
    end for
  end for
  // Update black nodes:
  for z = 1 to n_z − 1 do
    for y = 1 to n_y − 1 do
      for x = 1 + mod (z + y, 2) to n_x − 1 by 2 do
        update(x, y, z)
      end for
    end for
  end for
end for
```

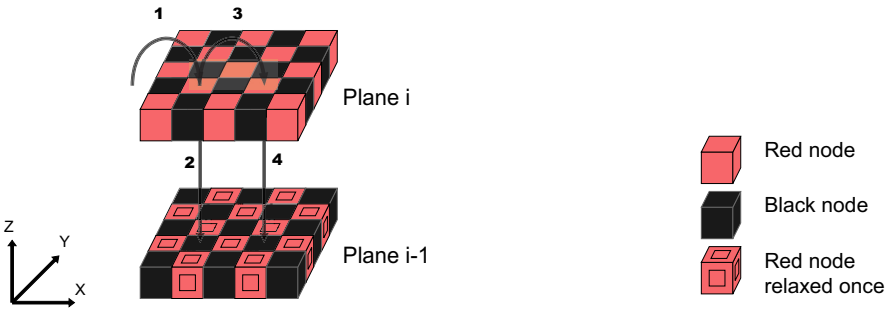


Figure 6.2: Loop fusion for red-black Gauss-Seidel in 3D.

Fusing the two original loop nests results in a single loop nest whose body exhibits a pairwise update pattern; as soon as the red node at position (x, y, z) has been updated, the black node at position $(x, y, z-1)$ can be updated since all data dependences are fulfilled. The corresponding pseudo-code is presented in Algorithm 6.4. Note that the red nodes located on the first interior plane of the grid as well as the black nodes located on the last interior plane of the grid must be updated separately. As a result of this transformation, the number of passes through the grid is reduced from $2i_{\max}$ to $i_{\max}$.

If both the unknowns belonging to four consecutive planes of the grid and the equation-specific data (i.e., the potentially variable stencil coefficients and the entries of the right-hand side) for the two current planes i and $i-1$ can be kept in cache, the resulting implementation requires the vector of unknowns to be loaded from main memory only once per iteration. This can be seen from

Algorithm 6.4 Red-black Gauss-Seidel in 3D after loop fusion.

```
1:  for i = 1 to i_max do
2:    // Update red nodes located on first interior grid plane (z = 1):
3:    ...
4:    // Update red nodes and black nodes in pairs:
5:    for z = 2 to n_z − 1 do
6:      for y = 1 to n_y − 1 do
7:        for x = 2 − mod (z + y, 2) to n_x − 1 by 2 do
8:          // Red node:
9:          update(x, y, z)
10:         // Black node:
11:         update(x, y, z − 1)
12:       end for
13:     end for
14:   end for
15:   // Update black nodes located on last interior grid plane (z = n_z − 1):
16:   ...
17: end for
```

Figure 6.2: in order that the fused implementation runs most efficiently, the unknowns on plane $i+1$ as well as the unknowns on plane $i-2$ must be kept in cache in addition to the unknowns and the equation-specific data corresponding to the current planes i and $i-1$.

After the application of loop fusion, loop distribution may be applied to the innermost loop (in dimension x) or to the loop in dimension y, see again Section 6.3.1. This would cause a line-wise or a plane-wise processing of red and black nodes, respectively.

Note that all further cache optimizations for red-black Gauss-Seidel presented in this work will assume that the sweeps over all red nodes and the sweeps over all black nodes have been fused into single sweeps over the entire grid. Consequently, loop fusion can be seen as an enabling transformation for our more involved blocking techniques (see Section 6.4).

6.3.3 Application of Loop Fusion to the LBM

A naive implementation of the LBM would perform two entire sweeps over the whole data set in every time step: one sweep for the collide operation, calculating the new distribution function values at each site, and a subsequent sweep for the stream operation, copying the distribution function values from each lattice site into its neighboring sites, see Section 4.2.

An elementary step to improve performance is to combine the collision and the streaming step. Analogous to the preceding case of red-black Gauss-Seidel, loop fusion can be introduced in order to replace the collide operation and the subsequent stream operation by a *fused collide-and-stream* step. Instead of passing through the data set twice per time step, the fused version calculates the new distribution function values at the current site and immediately copies the required data into the neighboring cells.

The loop fusion technique can also be applied if the stream step precedes the collide step, see Section 4.2.6. In this case, a *fused stream-and-collide* operations results. This situation has been illustrated in Figure 5.7, see Section 5.4.

All further cache optimizations for the LBM (i.e., all loop blocking techniques we will discuss in the following) assume that the two elementary operations of every time step (i.e., the streaming step and the collision step) have already been fused into a single update operation. As is the case for red-black Gauss-Seidel, loop fusion thus represents an enabling transformation for further cache optimizations for the LBM.

6.4 Loop Blocking

6.4.1 Motivation and Principle

Loop blocking (loop tiling) is a loop transformation which increases the depth of a loop nest with depth d by adding additional loops to the loop nest. The depth of the resulting loop nest will be anything from $d+1$ to $2d$. Loop blocking is primarily used to improve temporal locality by enhancing the reuse of data in cache and reducing the number of cache capacity misses (cf. Section 2.4.4), see [AK01, LRW91].

The introduction of loop blocking is illustrated by both parts of Algorithm 6.5, see also [KW03]. Assume that the code reads an array a with unit stride, whereas the access to array b is of stride n. Interchanging the loops will not help in this case since it would cause the array a to be accessed with a stride of n instead.

Algorithm 6.5 Loop blocking for matrix transposition.

```
1: // Original code:
2: for i = 1 to n do
3:    for j = 1 to n do
4:       a(i,j) = b(j,i);
5:    end for
6: end for
```

```
1: // Blocked code:
2: for ii = 1 to n by B do
3:    for jj = 1 to n by B do
4:       for i = ii to min(ii + B − 1, n) do
5:          for j = jj to min(jj + B − 1, n) do
6:             a(i,j) = b(j,i);
7:          end for
8:       end for
9:    end for
10: end for
```

Tiling a single loop replaces it by a pair of loops. The inner loop of the new loop nest traverses a *block (tile)* of the original iteration space with the same increment as the original loop. The outer loop traverses the original iteration space with an increment equal to the size of the block which is traversed by the inner loop. Thus, the outer loop feeds blocks of the whole iteration space to the inner loop which then executes them step by step. We generally use the term *k-way blocking* to indicate that k loops of the original loop nest have been blocked, causing the depth of the loop nest to be increased by k. The change in the iteration space traversal of the blocked loops in Algorithm 6.5 is shown in Figure 6.3, which is taken from [Wei01] and has been included in our joint publication [KW03] as well.

A very prominent example for the impact of the loop blocking transformation on data locality is matrix multiplication [BACD97, LRW91, WPD01], see also Section 9.2.2. In particular, the case of sparse matrices is considered in [NGDLPJ96]. The fundamental problem with computations involving unstructured sparse matrices (i.e., sparse matrices with an irregular sparsity pattern which occur in computations on unstructured grids, for example) is that, in contrast to their dense matrix counterparts, they imply overhead to accessing the index information and generally exploit less spatial and temporal locality. We also refer to Section 9.1.3.1, where data locality optimizations for numerical computations involving unstructured grids will be discussed.

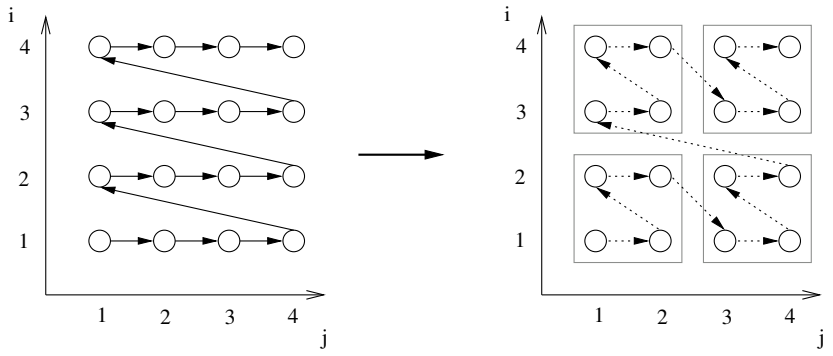


Figure 6.3: Iteration space traversal for original code and blocked code.

6.4.2 Loop Blocking in 2D

6.4.2.1 Blocked Red-Black Gauss-Seidel

Loop blocking approaches for red-black Gauss-Seidel in 2D have extensively been studied in [Wei01]. For further details, we refer to [KRWK00, WKKR99]. In particular, the case of variable coefficients, which additionally requires the use of appropriate cache-friendly data layouts (see Section 5.3), has been discussed in [Pfä01], see also [Dou96] and [KWR02].

A comparable blocking technique for Jacobi-type stencil-based iterative methods on regular 2D grids is presented in [BDQ98]. A blocked Gauss-Seidel algorithm based on structured 2D grids is described in [SC01]. This algorithm parallels the square blocking approach presented in [Wei01]. The application of a temporal locality metric to a standard version and a blocked version of the SOR method in 2D will be discussed in Section 9.4.3.2.

6.4.2.2 Blocking for CA and Jacobi-type Methods

We have developed two different blocking approaches for CA implementations in 2D; 1-way blocking and 3-way blocking. Again, we refer to the LBM as a representative example. See also [WPKR03a, WPKR03b, PKW$^+$03a, PKW$^+$03b]. In particular, [Wil03] contains comprehensive descriptions of these approaches, including many technical as well as problem-dependent details. As usual, these code transformations can also be applied to stencil-based Jacobi-type iterative schemes.

Algorithm 6.6 sketches the core of a standard implementation of the LBM in 2D. We assume that the array of cells is 0-based and that the two inner loops traverse all interior lattice sites, which are characterized by indices (x, y), where $1 \leq x \leq n_x$ and $1 \leq y \leq n_y$. Furthermore, we suppose that the update operation comprises both a stream and a collide operation, cf. Section 6.3.3, and that the actual layout of the data in memory is hidden within this update operation. We will refer to this basic formulation in the following.

1-way blocking. Figure 6.4 illustrates an example of 1-way blocking for the LBM in 2D. This technique parallels the *one-dimensional blocking* technique for red-black Gauss-Seidel in 2D, which is explained in detail in [Wei01]. Only interior grid cells are displayed. Moreover, this figure only shows the order of the cell updates, while the underlying data layout is excluded.

In the example shown in Figure 6.4, two consecutive time steps are blocked into a single pass through the grid. The areas surrounded by thick lines mark the blocks of cells to be updated next. White cells have been updated to time t, light gray cells to time $t + 1$, and dark gray cells to time $t + 2$. It can be seen that, in the second grid, all data dependences of the bottom row are fulfilled, and that it can therefore be updated to time $t + 2$, which is shown in the third grid. This is performed repeatedly until all cells have been updated to time $t + 2$.

Algorithm 6.6 Standard LBM in 2D.

```
1: for t = 1 to t_max do
2:   for y = 1 to n_y do
3:     for x = 1 to n_x do
4:       update(x, y)
5:     end for
6:   end for
7: end for
```

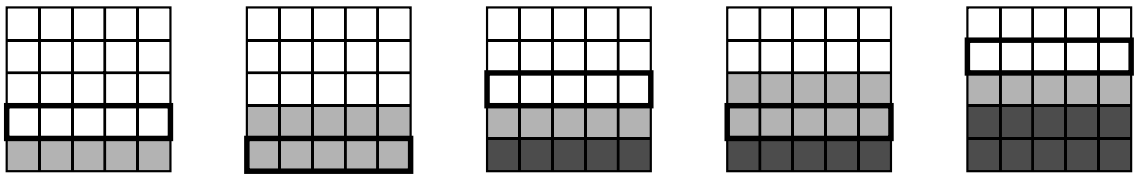


Figure 6.4: 1-way blocking technique for the 2D LBM.

Algorithm 6.7 is derived from Algorithm 6.6 and shows a general formulation of a 1-way blocked LBM in 2D. It assumes that t_B successive time steps are blocked into a single pass through the grid; i.e., the time loop has been blocked by t_B. For ease of presentation, we assume that t_B divides $t_{\max}$ (the total number of time steps to be executed).

This algorithm requires some special handling for the first $t_B - 1$ interior rows as well as for the last $t_B - 1$ interior rows of the grid. In a *preprocessing step* before the corresponding pass through the grid, the grid cells in rows y must be updated to time $tt + t_B - y - 1$, where $1 \leq y \leq t_B - 1$. Correspondingly, in a *postprocessing step* after the pass, the cells in rows y must be revisited until all of them have been updated to time $tt + t_B$, where $n_y - t_B + 2 \leq y \leq n_y$. Alternatively, the update step could implement an initial check in order to determine whether the node to be visited is an interior lattice site for which new distribution functions must be computed. Otherwise, the update operation could simply be skipped.

Note that this 1-way blocking technique works using either of the data layouts we have introduced in Chapter 5; e.g., grid merging and grid compression. For grid compression, in particular, this means that diagonal shifts and an alternating data placement pattern (which are necessary to maintain data dependences) must be introduced in addition to the data access pattern caused by the 1-way blocking transformation itself. Thus, data layout optimization and data access optimization can be considered orthogonal to each other.

If the grid compression technique is used to save memory and to enhance spatial locality, additional buffer space must be provided in order to maintain data dependences during the computation. We have already observed in Section 5.4 that, in the case of a CA in 2D, one additional row as well as one additional column must be allocated unless the time loop is blocked. If, however, the time

Algorithm 6.7 1-way blocking technique for the 2D LBM.

```
1:  for tt = 1 to t_max by t_B do
2:    // Preprocessing: special handling for rows 1 to t_B − 1:
3:    ···
4:    for y = t_B to n_y do
5:      for t = 0 to t_B − 1 do
6:        for x = 1 to n_x do
7:          update(x, y − t)
8:        end for
9:      end for
10:   end for
11:   // Postprocessing: special handling for rows n_y − t_B + 2 to n_y:
12:   ···
13: end for
```

loop is blocked using a block size of $t_B > 1$, t_B additional rows and t_B additional columns must be stored. This means that, for a CA with $n_x \times n_y$ cells, $(n_x + t_B)(n_y + t_B)$ cells need to be allocated in the case of grid compression, whereas the other data layouts require $2n_x n_y$ cells to be stored. We will illustrate this issue graphically in Section 6.4.3.2 in the context of data access optimizations for CA in 3D.

The downside to the 1-way blocking approach is that it only works efficiently if the data for at least $t_B + 2$ rows of lattice sites fits completely into cache. However, even two adjacent grid rows may already contain too much data to fit into cache, if the size of a grid row is too large. Therefore, the 1-way blocking approach will only yield performance speedups in the case of rather small grids, see Section 7.4. In order to overcome this problem, we have developed a more involved 3-way blocking approach, which will be demonstrated in the following.

3-way blocking. In Figure 6.5, we illustrate an example of 3-way blocking for the LBM in 2D, in which a 3×3 block of cells is employed. This technique parallels the *square two-dimensional blocking* approach for red-black Gauss-Seidel in 2D, which is discussed in detail in [Wei01]. As is the case for the 1-way blocking technique, our 3-way blocking approach works for either of the data layouts we have introduced in Chapter 5. Hence, Figure 6.5 only indicates the cell update pattern while ignoring the actual underlying data storage scheme. If, for example, the grid compression technique is used, the diagonal shifts introduced due to the 3-way blocking scheme must be performed in addition to the shifts due to the layout optimization technique itself. Again, the areas surrounded by thick lines mark the blocks of cells to be updated next.

In the example given in Figure 6.5, three successive time steps are performed during a single pass over the grid. Since the amount of data contained in the 2D block is independent of the grid size, it will always (if the block size is chosen appropriately) fit into cache. More precisely, it is necessary for the 3-way blocking approach to work efficiently that the cache, which the code is actually tailored for, can hold the data for $(x_B + t_B + 1)(y_B + t_B + 1)$ grid cells, where x_B, y_B, and t_B are the block sizes used for the loops along the spatial dimensions as well as for the time loop, respectively (cf. Algorithm 6.8).

In a D2Q9 lattice, a cell can be updated to time $t+1$ only if all eight adjacent cells correspond to time t. Consequently, the block to be processed moves diagonally down and left to avoid violating data dependences, see again Figure 6.5. Obviously, special handling is required for those sites near grid edges which cannot form a complete 3×3 block. In these cases, parts of the blocks outside the grid are simply chopped off.

Assume that we have arrived at the state of Grid 1. Grids 2 to 4 then demonstrate the handling of the next block of cells: Grid 2 shows that the marked block from Grid 1 has been updated to time $t + 1$, Grid 3 shows the update of the marked block from Grid 2 to time $t + 2$, and Grid 4 shows that the marked block from Grid 3 has been updated to time $t + 3$. Furthermore, the thick lines in Grid 4 mark the position of the 3×3 block to be updated next. Grids 5 to 7 illustrate the handling of this next block, and so on.

A general formulation of the 3-way blocked LBM in 2D is demonstrated in Algorithm 6.8. The body of the innermost loop is the *updateBlock()* operation, which traverses the current block of cells and updates all cells therein. As we have illustrated using the example from Figure 6.5, the position of the current block does not only depend on the indices xx and yy of the outer loops along the spatial grid dimensions, but also on the current time t. It is the influence of the last parameter t which causes the current block to move diagonally to the lower left in the course of each iteration of the t-loop.

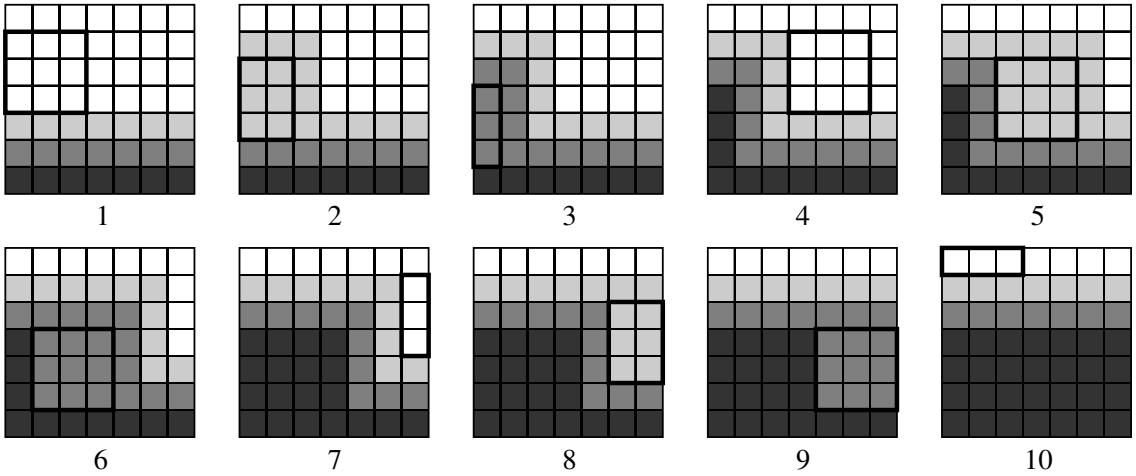


Figure 6.5: 3-way blocking technique for the 2D LBM.

Algorithm 6.8 3-way blocking technique for the 2D LBM.

```
1: for tt = 1 to t_max by t_B do
2:    for yy = 1 to n_y by y_B do
3:       for xx = 1 to n_x by x_B do
4:          for t = 0 to t_B − 1 do
5:             updateBlock(xx, yy, t)
6:          end for
7:       end for
8:    end for
9: end for
```

Note that the implementation of the *updateBlock()* operation itself is based on two nested loops which traverse the lattice sites inside the current block. Consequently, there are six nested loops (i.e., three additional loops), which explains the term 3-way blocking. For the sake of simplicity, we assume that the *updateBlock()* step checks if any cell to be updated is an interior lattice site and, if not, simply ignores it. Otherwise, appropriate handling for all cells which are located sufficiently close to the boundaries of the lattice could be implemented.

It is not obvious and hard to predict which of these two aforementioned implementations yields better performance results. The approach dispensing with conditional jump instructions in the body of the innermost loop may, on one hand, improve the effect of dynamic branch prediction [HP03]. On the other hand, it requires the implementation of special cases for cells that are located close to the grid boundaries. However, the special handling of these cells may cause a significant increase in code size, which in turn may imply performance drops due to limited instruction cache capacity. The question of how to treat cells or nodes located close to the boundaries represents a common code design problem which needs to be addressed as soon as loop blocking optimizations are applied to grid-based numerical methods.

If the underlying data layout is based on the grid compression technique, t_B additional rows and t_B additional columns must be allocated in order to preserve all data dependences of the algorithm. This exactly corresponds to what we have stated previously for the case of 1-way blocking.

6.4.3 Loop Blocking in 3D

6.4.3.1 Blocked Red-Black Gauss-Seidel

As is the case in 2D, a variety of different blocking techniques can be applied to red-black Gauss-Seidel in 3D. In the following, we will discuss approaches for 1-way, 2-way, 3-way, and 4-way blocking all of which are based on Algorithm 6.4. Recall that Algorithm 6.4 has been obtained by applying loop fusion to the standard version (Algorithm 6.3), cf. Section 6.3.2. The idea behind our blocking techniques is to successively increment the temporal locality of the code and therefore to enhance cache utilization.

Again, our blocking optimizations can be introduced as long as the discretizations of the differential operators yield 7-point stencils. See [KRTW02] and the more detailed discussion in [Thü02].

1-way blocking. Our 1-way blocking approaches are based on the idea of aggregating several consecutive iterations into a single pass through the grid. If the iteration loop (i.e., the i-loop in Algorithm 6.4) is blocked using block size i_B, then i_B successive iterations of red-black Gauss-Seidel will be lumped into a single pass.

From the viewpoint of linear algebra, this data access optimization can be interpreted as the successive application of (3.4), see Section 3.2.1.3. Using the notation we have introduced in Section 3.2.1.3, the $(k+m)$-th iterate $u^{(k+m)}$, $m \geq 1$, can be computed directly from the k-th iterate $u^{(k)}$, $k \geq 0$, as

$$u^{(k+m)} = M^m u^{(k)} + \left(\sum_{j=0}^{m-1} M^j \right) Nf \ , \tag{6.1}$$

see [Hac93].

A first example of this approach for the case $i_B = 2$ is illustrated in Figure 6.6. After a red node in plane i has been updated to iteration k (Step 1), the black node in plane $i-1$ that is located underneath this red node may immediately be updated to iteration k as well (Step 2). So far, this

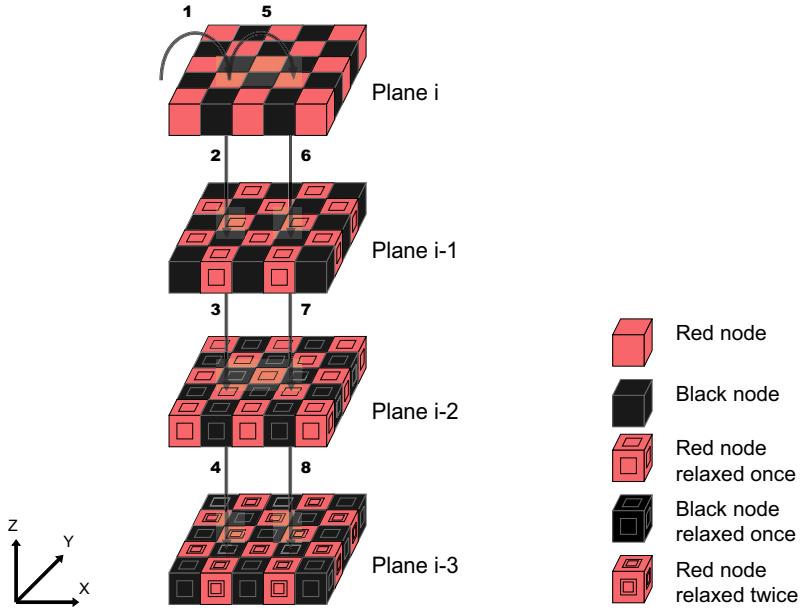


Figure 6.6: 1-way blocking technique for red-black Gauss-Seidel in 3D.

corresponds to the loop fusion technique we have introduced in Section 6.3.2, see also Figure 6.2. However, the 1-way blocking approach extends the loop fusion technique. After this black node in plane $i-1$ has been updated to iteration k, all six neighboring black nodes of the red node below (i.e., the marked red node located on plane $i-2$) are updated to iteration k and, hence, this red node can immediately be updated to iteration $k+1$ (Step 3). Consequently, the black node below (i.e., located on plane $i-3$) can also be updated to iteration $k+1$ (Step 4). Afterwards, the algorithm proceeds with the next red node on plane i (Step 5). It is easy to see that, in this example with $i_B = 2$, the lowermost $2i_B - 1 = 3$ grid planes and the uppermost $2i_B - 1 = 3$ grid planes need appropriate pre- and postprocessing, respectively.

The update pattern illustrated in Figure 6.6 results in two successive iterations (iterations k and $k+1$) being aggregated into a single pass through the grid. It is easy to see how this data access scheme can be modified to block more than two successive time steps; i.e., to handle the case $i_B > 2$. Correspondingly, in this more general situation, the lowermost $2i_B - 1$ interior grid planes and the uppermost $2i_B - 1$ interior grid planes require appropriate pre- and postprocessing, respectively.

A general version is shown in Algorithm 6.9. For ease of presentation, we assume that the block size i_B divides the total number of iterations $i_{\max}$ to be executed. Apparently, the application of our 1-way blocking scheme implies that only $i_{\max}/i_B$ instead of $i_{\max}$ sweeps over the grid need to be performed. Note that a sufficient condition for the code to run memory-efficiently is that the level of cache, which the code is tailored for, is large enough to store the unknowns on $2i_B + 2$ successive grid planes as well as the equation-specific data (i.e., the potentially variable stencil coefficients and the entries of the right-hand side) belonging to the current $2i_B$ grid planes.

Since the original loop nest has depth 4, blocking the iteration loop results in a loop nest of

Algorithm 6.9 1-way blocked red-black Gauss-Seidel in 3D, Version 1.

```
1:  for ii = 1 to i_max by i_B do
2:    // Preprocessing for the lowermost 2i_B − 1 interior grid planes (z = 1, ..., 2i_B − 1):
3:    ...
4:    // Traverse the grid nodes in a wavefront-like manner:
5:    for z = 2i_B to n_z − 1 do
6:      for y = 1 to n_y − 1 do
7:        for x = 2 − mod (z + y, 2) to n_x − 1 by 2 do
8:          for i = 0 to i_B do
9:            // Red node:
10:           update(x, y, z − 2i)
11:           // Black node:
12:           update(x, y, z − 2i − 1)
13:         end for
14:       end for
15:     end for
16:   end for
17:   // Postprocessing for the uppermost 2i_B − 1 interior grid planes (z = n_z − 2i_B + 1, ..., n_z − 1):
18:   ...
19: end for
```

depth 5. Further code transformations based on loop interchange and loop distribution can then be applied, cf. Sections 6.2 and 6.3. Depending on the ordering of the loops in this resulting loop nest, various update schemes are obtained.

A selection of possible access patterns resulting from different loop orders is illustrated in Figure 6.7. The left part of Figure 6.7 once more illustrates the 1-way blocking approach we have discussed previously, see Algorithm 6.9. Note that, for ease of presentation, the middle part and the right part of Figure 6.7 only illustrate the order in which the lines and the planes of the grid are visited, respectively, as well as the type of node (i.e., red or black) to be updated in the corresponding step.

The illustration in the middle of Figure 6.7 shows another data access pattern based on blocking the iteration loop. Here, the red nodes and the black nodes are updated in a line-wise manner, see Algorithm 6.10.

Finally, the right part of Figure 6.7 shows a data access pattern which exhibits a plane-wise update order of the red and the black nodes, respectively. This approach is presented in Algorithm 6.11. In Figures 6.6 and 6.7, the iterations loop has been blocked using a block size of $i_B = 2$.

It is hard to predict which of these approaches will yield the best performance. While these 1-way blocking schemes behave similarly in terms of cache utilization, different aspects of CPU architecture may significantly influence code execution. For example, if the innermost loop has a relatively long range, hardware-based as well as software-based data prefetching mechanisms can typically be applied very successfully [VL00] (see also Section 6.5.1), and the impact of pipeline stalls due to branch mispredictions is usually less dramatic [HP03]. On the other hand, if the innermost loop is comparatively short and its range is already known at compile time, the compiler might unroll this loop completely, thus eliminating loop overhead [GH01]. Often, manual loop

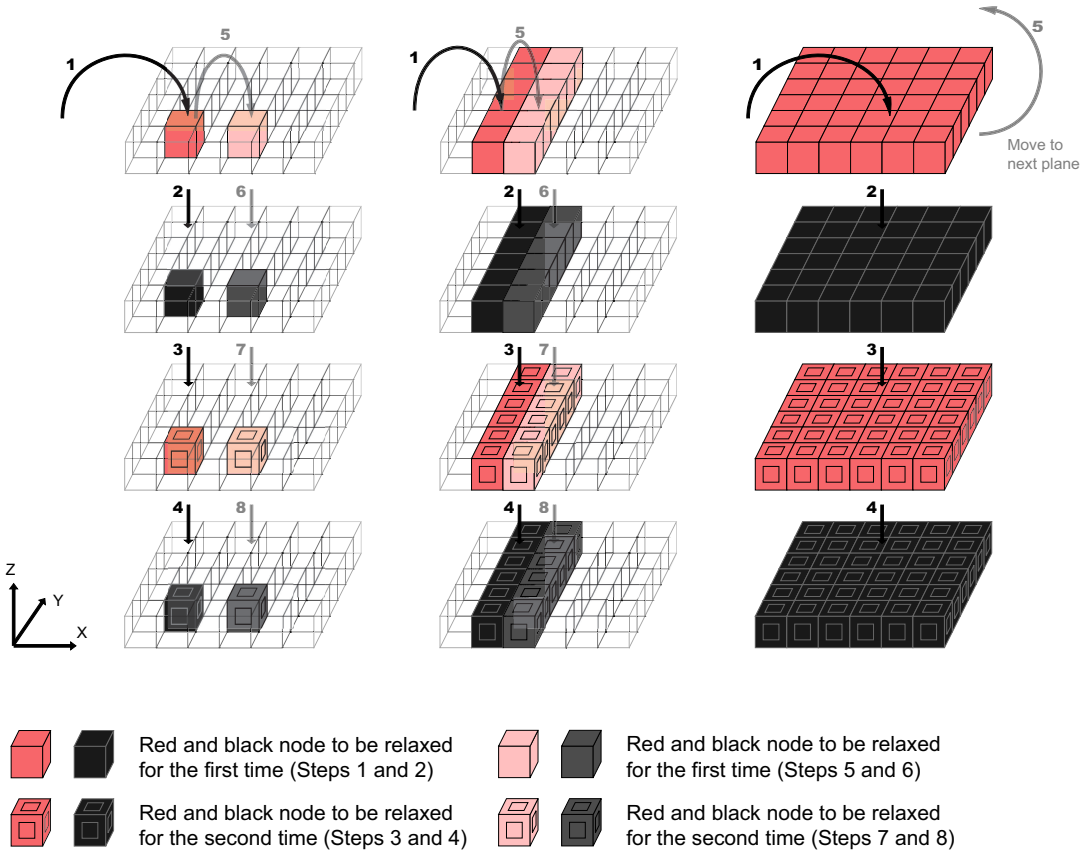


Figure 6.7: Different 1-way blocking approaches for red-black Gauss-Seidel in 3D.

Algorithm 6.10 1-way blocked red-black Gauss-Seidel in 3D, Version 2.

```
1: for ii = 1 to i_max by i_B do
2:   // Preprocessing for the lowermost 2i_B − 1 interior grid planes (z = 1, ..., 2i_B − 1):
3:   ...
4:   // Traverse the grid nodes in a wavefront-like manner:
5:   for z = 2i_B to n_z − 1 do
6:     for y = 1 to n_y − 1 do
7:       for i = 0 to i_B do
8:         // Red nodes:
9:         for x = 2 − mod (z + y, 2) to n_x − 1 by 2 do
10:          update(x, y, z − 2i)
11:        end for
12:        // Black nodes:
13:        for x = 2 − mod (z + y, 2) to n_x − 1 by 2 do
14:          update(x, y, z − 2i − 1)
15:        end for
16:      end for
17:    end for
18:  end for
19:  // Postprocessing for the uppermost 2i_B − 1 interior grid planes (z = n_z − 2i_B + 1, ..., n_z − 1):
20:  ...
21: end for
```

unrolling can also lead to an increase in performance, as will be demonstrated in Section 7.3.

2-way blocking. In addition to the iteration loop, there are three loops along the spatial dimensions which can be blocked in order to further increase the temporal locality of the code. Figure 6.8 illustrates a 2-way blocking approach where the loop in x-direction has been blocked as well.

Algorithm 6.12 shows this 2-way blocked version of red-black Gauss-Seidel. For the sake of simplicity, we assume that i_B again divides $i_{\max}$ and that x_B divides $n_x - 1$ (i.e., the total number of interior nodes to be updated in dimension x).

Analogous to the case of 1-way blocking, the application of other loop transformations such as loop interchange and loop distribution (cf. Sections 6.2 and 6.3) yields alternative 2-way blocked versions of red-black Gauss-Seidel. Figure 6.8 refers to the case $i_B = 2$ and $x_B = (n_x - 1)/2$. Note that, for this 2-way blocking scheme to work cache-efficiently, it is sufficient that the corresponding level of cache can hold the unknowns at approximately $3(x_B + 2)(2i_B + 2)$ grid nodes as well as the equation-specific data belonging to $2x_B i_B$ grid nodes.

3-way blocking. This approach results from blocking the iteration loop as well as the loops in dimensions x and y. In contrast to the 2-way blocking scheme we have presented previously, our 3-way blocking approach thus enables the reuse of neighboring data in dimension y as well. This 3-way blocking technique is characterized by a tile-wise data access pattern, where either the red nodes or the black nodes within the corresponding tile are updated, see Figure 6.9.

Algorithm 6.13 demonstrates the 3-way blocking technique for 3D red-black Gauss-Seidel. Again, for ease of presentation, we assume that i_B, x_B, and y_B divide the maximum values $i_{\max}$, $n_x - 1$,

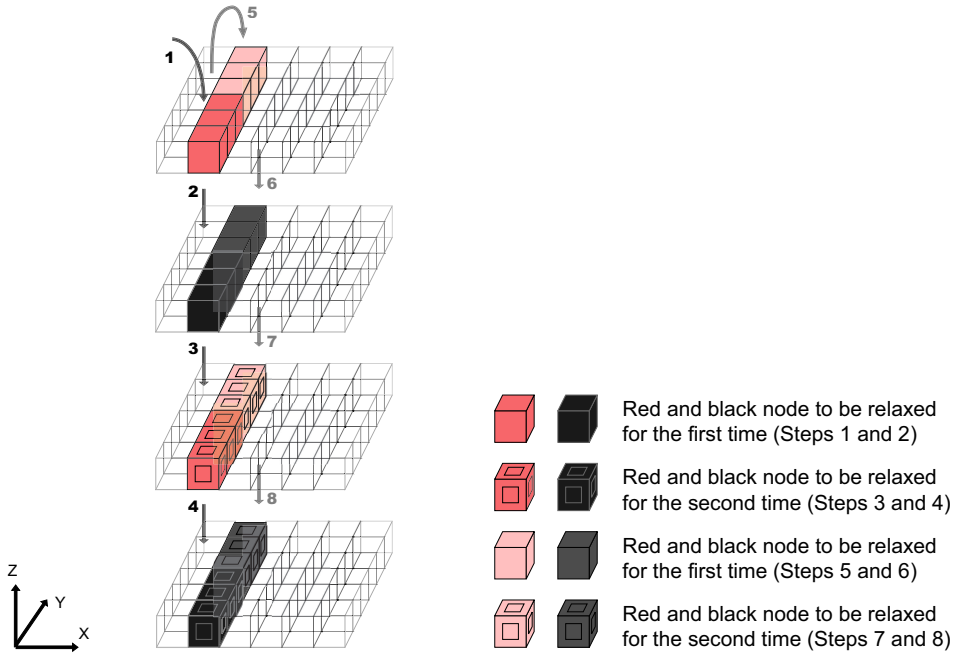


Figure 6.8: 2-way blocking technique for red-black Gauss-Seidel in 3D.

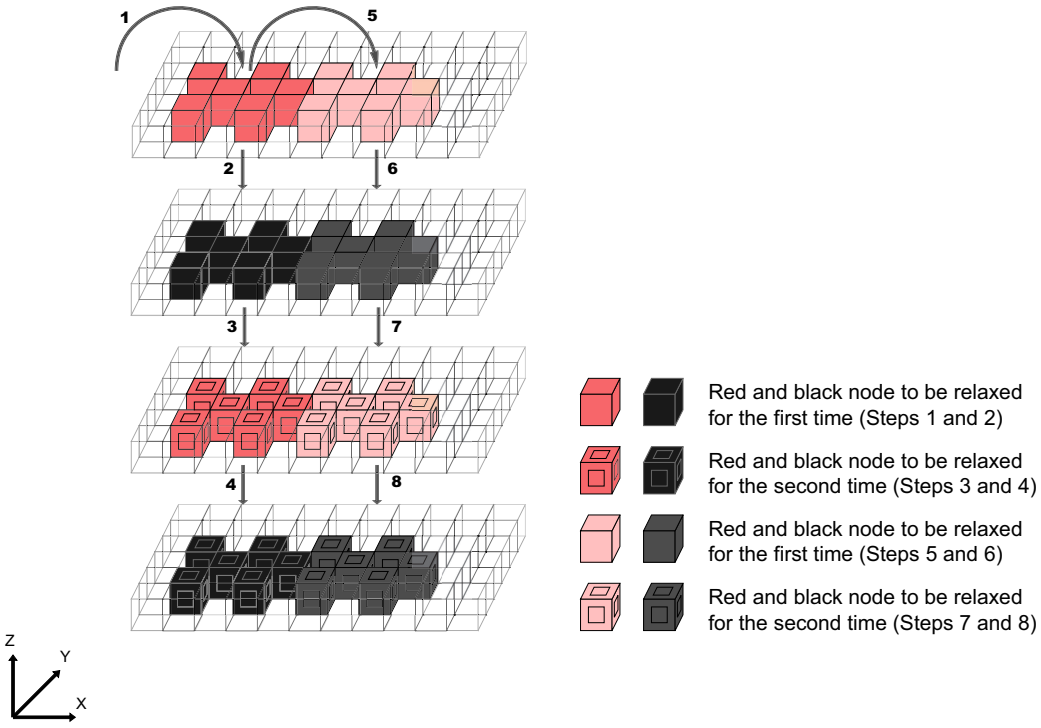


Figure 6.9: 3-way blocking technique for red-black Gauss-Seidel in 3D.

Algorithm 6.11 1-way blocked red-black Gauss-Seidel in 3D, Version 3.

```
1: for ii = 1 to i_max by i_B do
2:   // Preprocessing for the lowermost 2i_B − 1 interior grid planes (z = 1, ..., 2i_B − 1):
3:   ...
4:   // Traverse the grid nodes in a wavefront-like manner:
5:   for z = 2i_B to n_z − 1 do
6:     for i = 0 to i_B do
7:       // Red nodes:
8:       for y = 1 to n_y − 1 do
9:         for x = 2 − mod (z + y, 2) to n_x − 1 by 2 do
10:          update(x, y, z − 2i)
11:        end for
12:      end for
13:      // Black nodes:
14:      for y = 1 to n_y − 1 do
15:        for x = 2 − mod (z + y, 2) to n_x − 1 by 2 do
16:          update(x, y, z − 2i − 1)
17:        end for
18:      end for
19:    end for
20:  end for
21:  // Postprocessing for the uppermost 2i_B − 1 interior grid planes (z = n_z − 2i_B + 1, ..., n_z − 1):
22:  ...
23: end for
```

and $n_y - 1$ of the loop ranges, respectively. Furthermore, the two nested loops along the dimensions y and x which traverse the individual blocks (tiles) that are characterized by the current values of the loop indices yy and xx are hidden within the routines *updateRedNodesIn2DBlock()* and *updateBlackNodesIn2DBlock()*. Figure 6.9 refers to the case $i_B = 2$, $x_B = 4$, and $y_B = 3$. In order that our 3-way blocked red-black Gauss-Seidel code performs efficiently, it is sufficient that the cache can keep the unknowns at $(x_B + 2)(y_B + 2)(2i_B + 2)$ grid nodes as well as the equation-specific data at $2x_B y_B i_B$ grid nodes.

4-way blocking. As the name suggests, all four loops of the original loop nest (cf. Algorithm 6.4) have been blocked in our 4-way blocked version of red-black Gauss-Seidel in order to further enhance temporal locality. This results in an update pattern which is characterized by a cuboid-shaped block traversing the full 3D grid. During each iteration of the ii-loop, the block moves diagonally in direction $(-1, -1, -1)^T$ in order to maintain all data dependences.

This cache optimization scheme therefore parallels the square blocking technique for red-black Gauss-Seidel in 2D [Wei01] as well as the 3-way blocking approach for the 2D LBM from Section 6.4.2.2. Correspondingly, special handling is required for those blocks near grid edges. In these cases, parts of the blocks outside the grid are simply chopped off. A similar blocking technique for the LBM in 3D will be introduced below in Section 6.4.3.2.

Figure 6.10 illustrates the movement of a $4 \times 4 \times 4$ block within one iteration of the ii-loop. The color of each block denotes the type of the grid nodes to be updated within the block (i.e.,

Algorithm 6.12 2-way blocked red-black Gauss-Seidel in 3D.

```
1: for ii = 1 to i_max by i_B do
2:   // Preprocessing for the lowermost 2i_B − 1 interior grid planes (z = 1, ..., 2i_B − 1):
3:   ···
4:   // Traverse the grid nodes in a wavefront-like manner:
5:   for z = 2i_B to n_z − 1 do
6:     for y = 1 to n_y − 1 do
7:       for xx = 1 to n_x − 1 by x_B do
8:         for i = 0 to i_B do
9:           // Red nodes:
10:          for x = xx + mod (z + y + xx, 2) to xx + x_B by 2 do
11:            update(x, y, z − 2i)
12:          end for
13:          // Black nodes:
14:          for x = xx + mod (z + y + xx, 2) to xx + x_B by 2 do
15:            update(x, y, z − 2i − 1)
16:          end for
17:        end for
18:      end for
19:    end for
20:  end for
21:  // Postprocessing for the uppermost 2i_B − 1 interior grid planes (z = n_z − 2i_B + 1, ..., n_z − 1):
22:  ···
23: end for
```

Algorithm 6.13 3-way blocked red-black Gauss-Seidel in 3D.

```
1: for ii = 1 to i_max by i_B do
2:   // Preprocessing for the lowermost 2i_B − 1 interior grid planes (z = 1, ..., 2i_B − 1):
3:   ···
4:   // Traverse the grid nodes in a wavefront-like manner:
5:   for z = 2i_B to n_z − 1 do
6:     for yy = 1 to n_y − 1 by y_B do
7:       for xx = 1 to n_x − 1 by x_B do
8:         for i = 0 to i_B do
9:           updateRedNodesIn2DBlock(xx, yy, z − 2i)
10:          updateBlackNodesIn2DBlock(xx, yy, z − 2i − 1)
11:        end for
12:      end for
13:    end for
14:  end for
15:  // Postprocessing for the uppermost 2i_B − 1 interior grid planes (z = n_z − 2i_B + 1, ..., n_z − 1):
16:  ···
17: end for
```

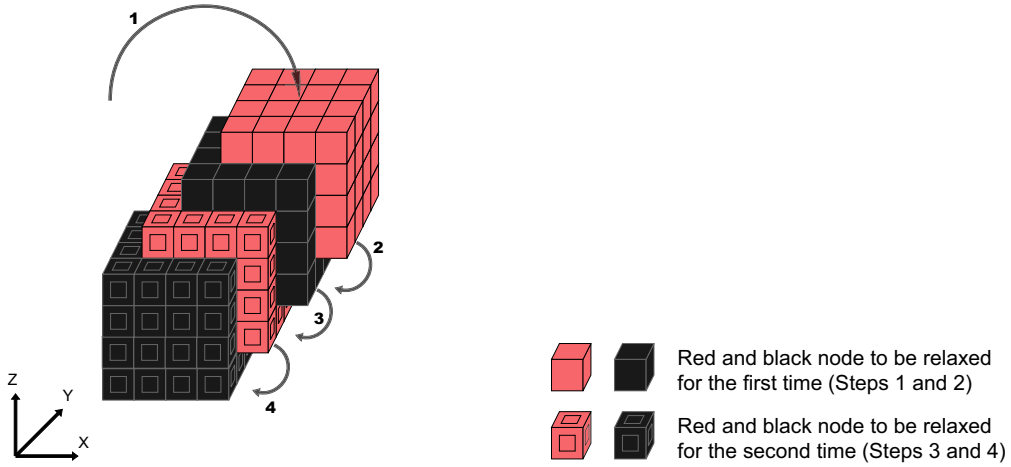


Figure 6.10: 4-way blocking technique for red-black Gauss-Seidel in 3D.

red or black). In this example, we have chosen $i_B = 2$; i.e., two consecutive iterations of red-black Gauss-Seidel have been blocked into a single pass through the full grid. The loops along the spatial dimensions have been blocked using the block sizes $x_B = y_B = z_B = 4$, hence the $4 \times 4 \times 4$ blocks.

The pseudo-code for our 4-way blocking technique is shown in Algorithm 6.14. Analogous to the previous case of 3-way blocking, the innermost loops which traverse the current block (characterized by the loop indices zz, yy, and xx) are hidden within the routines *updateRedNodesIn3DBlock()* and *updateBlackNodesIn3DBlock()*. In addition, the diagonal displacement of the blocks is hidden by the distinction of the different names of these routines as well as by the the fourth parameter that specifies the current index of the i-loop.

In order that our 4-way blocked red-black Gauss-Seidel code performs memory-efficiently, it is sufficient that the corresponding level of cache is large enough to store the components of the vector of unknowns at $(x_B + i_B + 2)(y_B + i_B + 2)(z_B + i_B + 2)$ grid nodes as well as the equation-specific

Algorithm 6.14 4-way blocked red-black Gauss-Seidel in 3D.

```
1:  for ii = 1 to i_max by i_B do
2:    for zz = 1 to n_z − 1 + z_B by z_B do
3:      for yy = 1 to n_y − 1 + y_B by y_B do
4:        for xx = 1 to n_x − 1 + x_B by x_B do
5:          for i = 0 to i_B do
6:            updateRedNodesIn3DBlock(xx, yy, zz, i)
7:            updateBlackNodesIn3DBlock(xx, yy, zz, i)
8:          end for
9:        end for
10:     end for
11:   end for
12: end for
```

data at $(x_B + i_B)(y_B + i_B)(z_B + i_B)$ grid nodes, cf. Figure 6.10. It is obvious that a larger 3D block implies a higher reuse of cache contents as long as the cache capacity is not exceeded and, moreover, cache conflict misses are eliminated using appropriate data layout optimization techniques.

Note that the upper bounds for the outer loops along the spatial dimensions (i.e., the xx-loop, the yy-loop, and the zz-loop) have been increased by the corresponding block sizes x_B, y_B, and z_B, respectively, to ensure that all grid nodes are updated correctly. Obviously, special handling is required for those nodes located near the grid boundary where they cannot be included in complete $x_B \times y_B \times z_B$ blocks. In these cases, parts of the blocks outside the grid are simply chopped off by adjusting loop parameters. Again, this is hidden within the operations *updateRedNodesIn3DBlock()* and *updateBlackNodesIn3DBlock()*. Similar to the case of 3-way blocking for CA (see Section 6.4.2.2), alternative approaches for the update of the nodes near the boundary are possible.

6.4.3.2 Blocking for CA and Jacobi-type Methods

In this section, we present a loop blocking technique, which we have developed in order to enhance the cache performance of CA as well as stencil-based Jacobi-type methods in 3D. Again, we focus on the LBM, which was introduced in Section 4.2. The following illustrations are taken from [Igl03].

While a 3-way blocking approach has been discussed in [PKW$^+$03a, PKW$^+$03b] and, in much more detail, in [Igl03], we restrict ourselves to the explanation of our 4-way blocking technique. See also the comprehensive discussion which is provided in [Igl03]. The experimental results we will provide in Section 7.5 reveal that the 4-way blocked LBM codes generally outperform their 3-way blocked counterparts.

As was observed in Section 4.1, CA parallel Jacobi-type stencil-based iterative schemes; each grid cell is updated using its own state and the states of the cells in its neighborhood at the corresponding previous discrete point in time. Hence, the update must not be done in place since data dependences need to be maintained. See also the discussion of optimizing data layout transformations for CA in Sections 5.3.3 and 5.4.

Data access pattern. Figures 6.11 and 6.12 illustrate our 4-way blocking approach. In order to simplify the presentation, we use a $3 \times 3 \times 3$ block of lattice sites, while typical block sizes would be larger and depend on the cache capacity. Three successive time steps — indicated by three different colors — are blocked into a single pass through the lattice. Note that, for clarity reasons, these two figures only picture the data access pattern while ignoring the actual data layout on which the implementation is based. Analogous to the 2D case, the data access pattern and the underlying data layout can be considered orthogonal to each other.

Algorithm 6.15 provides an abstract formulation of our 4-way blocking approach. We assume that the interior grid cells to be updated in the course of the computation are located at positions represented by array indices (x, y, z), where $1 \leq x \leq n_x$, $1 \leq y \leq n_y$, and $1 \leq z \leq n_z$, cf. the 2D case presented in Section 6.4.2.2. The correct placement of the current $x_B \times y_B \times z_B$ block of cells is hidden in the *updateBlock()* routine. Analogous to the cases of 3-way blocking for the LBM in 2D (see Section 6.4.2.2) and 4-way blocking for red-black Gauss-Seidel in 3D (see Section 6.4.3.1), those parts of the current block outside the actual grid (i.e., outside the range of array indices specified previously) are simply chopped off. However, there are several alternative approaches for handling the grid nodes near the boundaries.

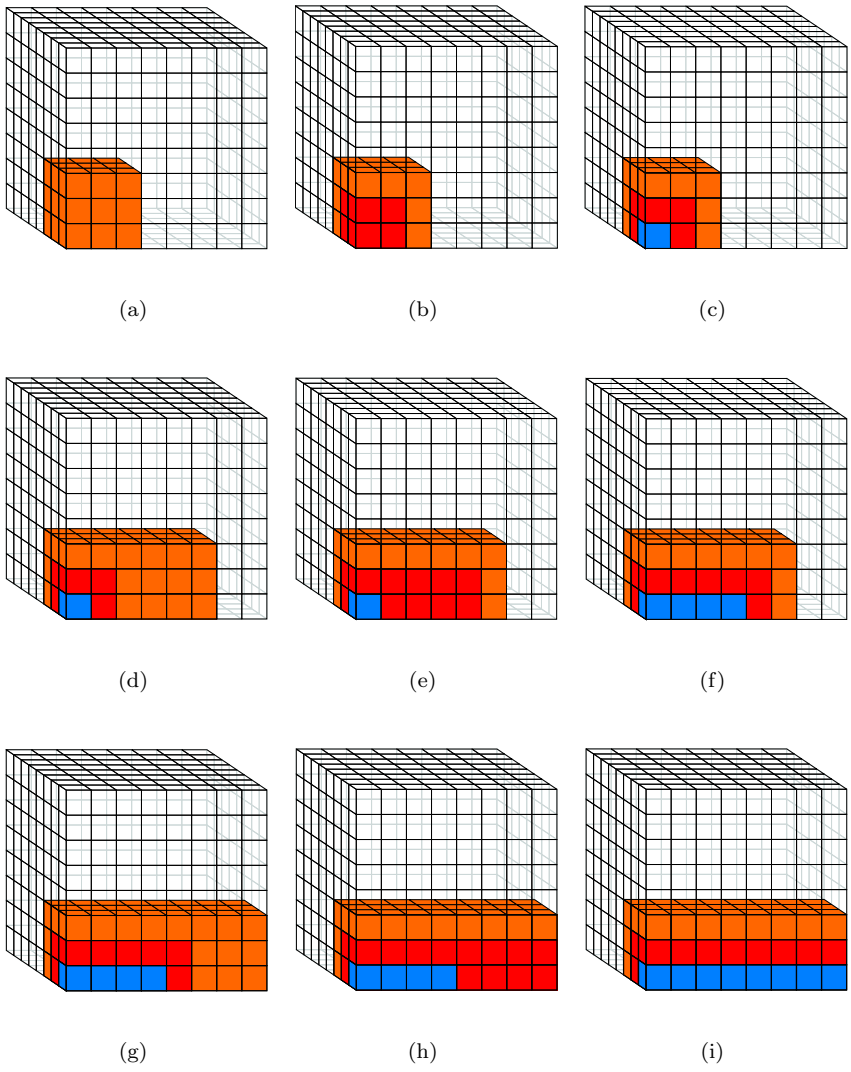


Figure 6.11: Update pattern 4-way blocking technique for the 3D LBM.

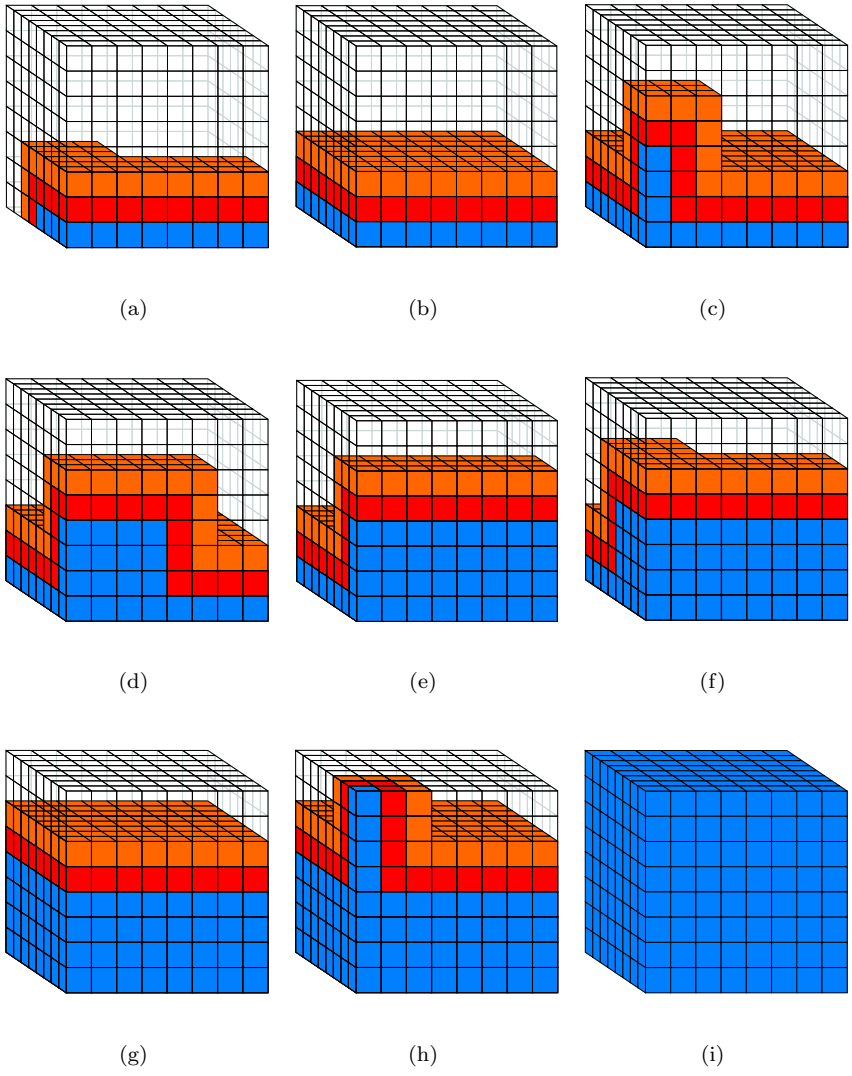


Figure 6.12: Update pattern 4-way blocking technique for the 3D LBM (continued).

Algorithm 6.15 4-way blocking technique for the 3D LBM.

```
1:  for tt = 1 to t_max by t_B do
2:    for zz = 1 to n_z by z_B do
3:      for yy = 1 to n_y by y_B do
4:        for xx = 1 to n_x by x_B do
5:          for t = 0 to t_B − 1 do
6:            updateBlock(xx, yy, zz, t)
7:          end for
8:        end for
9:      end for
10:   end for
11: end for
```

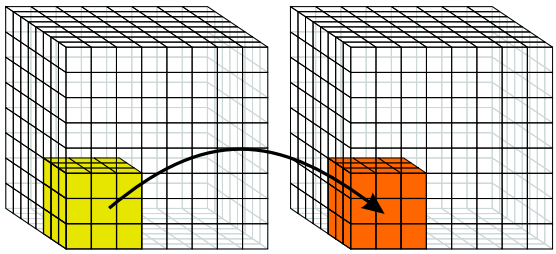
Data layout based on two separate grids. Figure 6.13 provides a more detailed picture of our 4-way blocking technique. In this case, the implementation is based on a standard data layout that is characterized by two separate grids allocated in memory; i.e., a source grid and a destination grid, which change roles after each time step. In this setting, $2n_xn_yn_z$ cells need to be stored, which equals the data size of two full grids. The large arrows indicate the current source and destination grid.

Again, $3 \times 3 \times 3$ blocks are employed, and three successive time steps are blocked into a single pass though the data set. Using the identifiers from Algorithm 6.15, Figure 6.13 thus refers to the situation where $x_B = y_B = z_B = t_B = 3$. For ease of presentation, only the first few steps are depicted in Figure 6.13. Note that array merging has previously been applied once in order to aggregate the distribution functions of the individual grid cells, see Section 5.3.3.

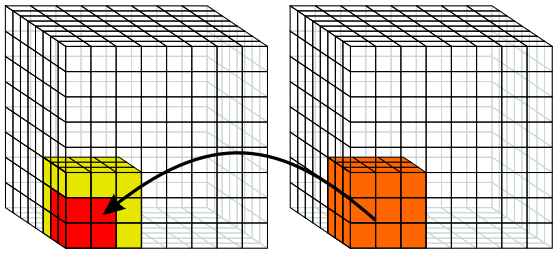
Data layout based on grid compression. Alternatively, the grid compression technique can be applied in order to save memory and to enhance data locality. This situation is illustrated in detail in Figures 6.14 and 6.15. As was explained in Section 5.4, it is sufficient to allocate a single lattice in memory instead of both a full source grid and a full destination grid, as long as additional buffer space is provided to avoid overwriting data that will still be needed for future cell updates. This data layout based on grid compression requires only $(n_x + t_B)(n_y + t_B)(n_z + t_B)$ grid cells to be allocated in memory.

Figure 6.14 shows the first steps of our 4-way blocked implementation of the 3D LBM. Note that the data access pattern arising from this blocking approach (see Figures 6.11 and 6.12) occurs in addition to the diagonal shifts which stem from the application of the layout optimization by grid compression. Again, this figure refers to the case where $x_B = y_B = z_B = t_B = 3$. Those parts of the moving $3 \times 3 \times 3$ block which are located outside the lattice are assumed to be chopped off. Hence, the moving block of lattice sites to be updated becomes smaller, see Parts (c) and (d) of Figure 6.14.

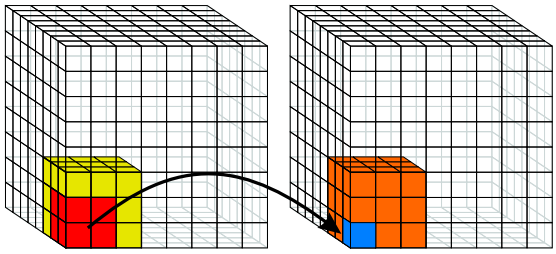
Figure 6.15 illustrates the arrangement of the data in memory after a first pass through the grid; i.e., after performing the first $t_B = 3$ time steps. The initial grid has moved three steps down along the diagonal. Afterwards, in the course of the execution of the next $t_B = 3$ time steps, the diagonal shifts will move the grid back along the diagonal into its original position.



(a)

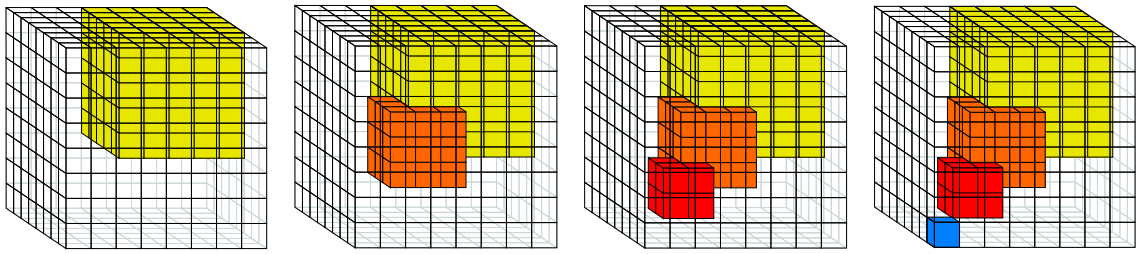


(b)



(c)

Figure 6.13: 4-way blocking technique for the 3D LBM based on two separate grids.



(a) (b) (c) (d)

Figure 6.14: Initial steps of the 4-way blocking technique for the 3D LBM based on grid compression.

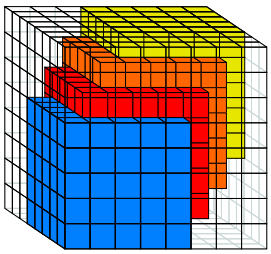


Figure 6.15: Memory layout of the 4-way blocking technique for the 3D LBM based on grid compression after one single pass through the grid.

6.5 Further Data Access Optimizations

6.5.1 Data Prefetching

The loop transformations discussed so far aim at reducing the number of capacity misses which occur in the course of a computation. Misses which are introduced by first-time accesses (cold-start misses, see Section 2.4.4) are not addressed by these optimizations. *Data prefetching* allows the microprocessor to issue a data request before the computation actually requires the data. If the data is requested early enough, the penalty of both cold-start misses and capacity misses not covered by loop transformations can be hidden [VL00].

Many of today's microprocessors implement a *prefetch* machine instruction which is issued as a regular instruction. The prefetch instruction is similar to a load, with the exception that the data is not forwarded to the CPU after it has been cached. Prefetch instructions can be inserted into the code manually by the (assembler) programmer or automatically by a compiler. In both cases, prefetching involves overhead. The prefetch instructions themselves have to be executed; i.e., pipeline slots will be filled with prefetch instructions instead of other instructions ready to be executed. Furthermore, the memory addresses of the prefetched data items must be calculated and will be calculated again when the load operations, which actually fetch the data items from the cache into the CPU, are executed.

In addition to software-based prefetching, hardware schemes have been proposed and implemented as well. They add prefetching capability to a system, eliminating the need for prefetch instructions. One of the simplest hardware-based prefetching schemes is *sequential prefetching*; whenever a cache block is accessed, a certain number of subsequent cache blocks are prefetched as

well. Note that this approach was proposed more than 20 years ago [Smi82].

More sophisticated prefetch schemes using reference prediction tables have been invented since then [CB95]. Other hardware-based prefetching mechanisms are based on dedicated FIFO memories, so-called *stream buffers* [PK94]. Simulation results concerning the impact of sophisticated hardware prefetching techniques on cache miss penalties are discussed in [LRB01]. A cooperative hardware/software prefetching scheme which combines the accuracy of software prefetching with the efficiency of aggressive hardware prefetching has been presented in [WBR+03].

Note that prefetching can only be successful if the data stream is predicted correctly either by the hardware or by the compiler, and if the cache is large enough to keep the prefetched data together with currently needed memory references. If the prefetched data replaces data which is still needed, this will lead to an increase in bus utilization, overall cache miss rates, as well as effective memory latencies. Again, we refer to [VL00] for a comprehensive survey of data prefetching mechanisms.

Finally, it is important to point out that, while data prefetching techniques can reduce effective access latencies, they cannot mitigate the effects due to inadequate memory bandwidth. However, it is often the limited bandwidth of the memory subsystem which is responsible for the poor performance of grid-based numerical computations on which we focus in this thesis.

6.5.2 Data Copying

In Section 6.4, loop blocking was introduced as a technique to reduce the number of capacity misses. Unfortunately, blocked codes often suffer from a high degree of conflict misses introduced by self-interference [WL91]. This effect is demonstrated by means of Figure 6.16, which is taken from [Wei01]. The figure shows a part (block) of a big array which is to be reused by a blocked algorithm. Suppose that a direct mapped cache is used, and that the two words marked with x are mapped to the same cache location. Due to the regularity of the cache mapping, the shaded words in the upper part of the block will be mapped to the same cache frames as the shaded words in the lower part of the block. Consequently, if the block is accessed repeatedly, the data in the upper left corner will replace the data in the lower right corner and vice versa, thus reducing the reusable part of the block.

Therefore, a data copying technique to guarantee high cache utilization for blocked algorithms has been proposed [WL91]. With this approach, noncontiguous data from a logical block is copied into a contiguous area of memory. Hence, each word of the block will be mapped to its own cache location, effectively avoiding self-interference within the block.

The technique, however, involves a copy operation which increases the total cost of the algorithm. In many cases, the additional cost will outweigh the benefits from copying the data. Hence, a compile time strategy has been introduced in order to determine when to copy data [TGJ93]. This technique is based on an analysis of cache conflicts.

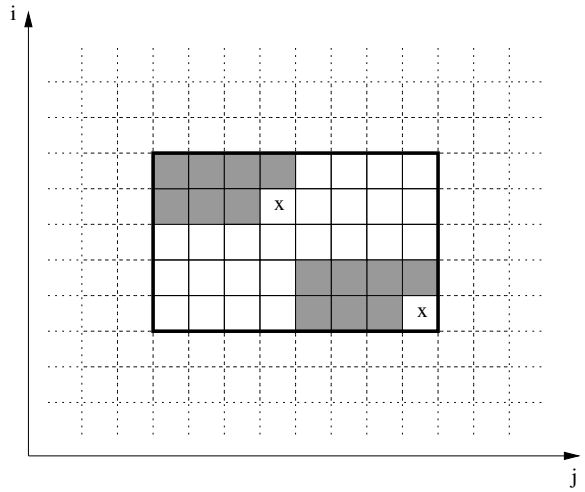


Figure 6.16: Self-interference in blocked code.

Chapter 7

Experimental Results

7.1 Initial Remarks

7.1.1 Combining Efficiency and Flexibility

On one hand, in order to obtain reasonable performance due to high utilization of available hardware resources, programs must respect the specific properties of the underlying machine architectures. This is particularly true for numerically intensive codes whose execution does not suffer from comparatively slow I/O traffic or even from user interaction.

On the other hand, programs should be flexible and portable such that they can be run efficiently on a potentially large set of target platforms. These platforms may consist of different CPU types, different interconnection networks, or different memory architectures.

Therefore, it is a general challenge in software engineering to design and to implement numerically intensive code which is both *efficient* as well as *portable*. This is already true — and a fundamental problem — for the restricted case of target platforms that are characterized by cache architectures with different properties; e.g., capacity and associativity.

We have designed and implemented all our codes such that any data layout optimization from Chapter 5 can easily be combined with any data access optimization from Chapter 6. This has primarily been accomplished by a highly modular code structure and the extensive use of macro preprocessing techniques. Alternatively, if a suitable programming language is used, object-oriented programming techniques can be applied. However, the latter must be introduced very carefully in order to avoid overhead at runtime and to ensure efficient code execution [BM99].

In the case of the cache-efficient implementation of iterative linear solvers, different data layouts for the matrices and the vectors are implemented in different header files, cf. our layout schemes in Section 5.3.2. Each of these header files implements the same set of preprocessor macros such that the actual implementation and the subsequent optimization of the data access patterns becomes independent of the choice of the data layout.

Our cache-optimized LBM programs are organized very similarly. In particular, the introduction of the alternative data access policies, which we will explain in Section 7.5.1 below, is based on the use of different header files. Each header file implements both the stream-and-collide update step and the collide-and-stream update step using preprocessor macros. As a consequence, the source files only need to contain code skeletons which implement the loop structures resulting from the application of the various data access optimizations from Sections 6.4.2.2 and 6.4.3.2. The bodies of the corresponding innermost loops consist of the previously mentioned preprocessor macros for

the update steps. Note that this macro-oriented approach can also be used in the case of the grid compression scheme where the directions of the update sweeps over the grid alternate, see Sections 5.4, 6.4.2.2, and 6.4.3.2. In addition to the enhanced flexibility of the code, another advantage of this programming approach is that the source files can be kept quite short and, moreover, be structured very clearly.

Our codes follow the AEOS (automated empirical optimization of software) paradigm [WPD01]; i.e., their tuning process is based on appropriately searching the parameter space in order to determine the most efficient program configuration. The codes are instantiated at compile time by specifying a number of hardware-dependent optimization parameters; e.g., block sizes and array paddings. While some parts of the compiler community might consider this approach as inelegant, it has proven to yield excellent results in terms of code efficiency. An overview of automatic performance tuning will be given in Section 9.1.2.

7.1.2 Purpose of this Chapter

Our performance results show that the application of the cache optimization techniques we have introduced in Chapters 5 and 6 can significantly enhance the efficiency of our target implementations. However, note that we do *not* claim at all that our performance results are *optimal* for any of the platforms under consideration. Rather, we intend to emphasize the universal applicability and the generality of our optimizing code transformations. An overview of the set of platforms we have used can be found in Appendix A.

7.2 Iterative Methods on 2D Grids

Optimization techniques to enhance the cache performance of iterative methods in 2D have extensively been studied in [Wei01] and, for the case of variable coefficients, in [Pfä01]. These references also contain detailed discussions of performance results.

Hence, we omit the presentation of performance results for cache-optimized iterative solvers in 2D and refer to the previously listed literature as well as to our joint publications, especially to [KRWK00, WKKR99, DHK+00b, DHI+00, KWR02].

7.3 Iterative Methods on 3D Grids

7.3.1 Description of the Implementations

All codes we discuss in this section have been written in Fortran77. All floating-point numbers are stored as double precision values. Our implementations are based on regular 3D grids and assume a matrix involving (constant or variable) 7-point stencils. Hence, the codes can in principle be used to handle scalar elliptic differential equations such as $Lu := -\nabla \cdot (a \nabla u) = f$ on any cubic domain. Here, $a = a(x, y, z)$ is a positive scalar function that denotes some appropriate variable conductivity or diffusion parameter. The scalar function $f = f(x, y, z)$ represents the right-hand side. Furthermore, we assume Dirichlet boundary conditions only.

Since we are only interested in the performance and particularly in the memory access behavior of the codes, the actual choice of the differential operator is irrelevant. We have therefore chosen the special case of the negative Laplace operator $L := -\Delta u$ — see Section 3.3.2 — and the corresponding standard second-order discretization based on finite differences. For the case of variable coefficients,

we have included into our codes the alternative data layout schemes from Section 5.3.2; i.e., the band-wise, the access-oriented, and the equation-oriented data layout.

We have implemented the following numerically equivalent versions of the red-black Gauss-Seidel method:

- *Standard w/o padding:*
  This code corresponds to the standard implementation shown in Algorithm 6.3 in Section 6.3.2.

- *Standard w/ padding:*
  As the name suggests, nonstandard intra-array padding has been introduced into the standard implementation, see Section 5.2.3.3, Algorithm 5.7. This type of array padding is used for all subsequent code versions.

- *Fused red-black Gauss-Seidel:*
  According to Algorithm 6.4 in Section 6.3.2, the two successive sweeps over the red nodes and the black nodes have been fused into a single pass through the data set.

- *1-way blocking:*
  This implementation corresponds to the 1-way blocking approach that has been presented in Figure 6.6 as well as in Algorithm 6.9 in Section 6.4.3.1.

- *1-way blocking with loop interchange:*
  This 1-way blocked code corresponds to the line-wise update pattern we have illustrated in the middle part of Figure 6.7 as well as in Algorithm 6.10 in Section 6.4.3.1.

- *1-way blocking, hand-optimized:*
  This code has been derived from the aforementioned *1-way blocking* approach. The innermost loop (i.e., the i-loop in Algorithm 6.9) has been unrolled manually.

- *2-way blocking:*
  This version implements the 2-way blocking approach which we have illustrated in Figure 6.8 and formulated in Algorithm 6.12, see again Section 6.4.3.1.

- *3-way blocking:*
  This code corresponds to the 3-way blocking approach for red-black Gauss-Seidel. We refer to Figure 6.9 and Algorithm 6.13 in Section 6.4.3.1.

- *4-way blocking:*
  Finally, our 4-way blocked code implements the cache optimization technique we have demonstrated in Figure 6.10 and further sketched in Algorithm 6.14, see again Section 6.4.3.1.

These implementations of the red-black Gauss-Seidel method have been integrated into a multigrid framework based on a V-cycle coarse-grid correction scheme. For details, we refer to Section 3.3 and to the multigrid literature we cite therein. The grid hierarchy consists of structured grids. Its finest level contains $2^n + 1$, $n > 0$, nodes in each dimension. We assume the simplest case of Dirichlet boundaries which we store explicitly in order that each interior node can be assigned a 7-point stencil[1]. Consequently, on the finest level of the grid hierarchy, there are $(2^n - 1)$ interior nodes (i.e., unknowns) per dimension and $(2^n - 1)^3$ unknowns in total. Hence, the corresponding

[1]This simplifies the implementation since no special handling is required for grid nodes located close to the boundary.

matrix A has $(2^n - 1)^3$ rows and the same number of columns. In the performance figures below, the labels of the x-axis refer to the numbers $2^n + 1$ of grid nodes per dimension.

The coarser levels of the hierarchy are obtained by recursively applying standard coarsening. We discretize the continuous operator on each grid level anew such that the band structure of the resulting matrices does not change from level to level. The coarsest level of a hierarchy contains only a single unknown such that one Gauss-Seidel iteration is enough to solve the corresponding linear equation in each multigrid V-cycle exactly. The optimized smoothers we have introduced previously are only applied on those grids of the hierarchy that are sufficiently large. On the small grids, we have always used a standard implementation of the red-black Gauss-Seidel smoother. According to the results in Section 7.3.2.1, this influences the overall performance of the multigrid codes only marginally.

As a consequence, this multigrid setting implies that, for the individual grid level with $2^n = 128$, about 147 MB of memory are required to store the respective matrix (i.e., the stencil weights), the unknown vector, and the right-hand side. In addition, about 22 MB are necessary to store the data corresponding to the seven coarser grid levels.

We have chosen full weighting and trilinear interpolation as inter-grid transfer operators. Since the discrete operators are based on 7-point stencils, any red node only depends on black nodes (or on fixed boundary nodes), and vice versa; see again Section 3.2.3. Hence, after a red-black Gauss-Seidel iteration, the residuals at the black nodes vanish and, in the course of the residual restriction operation, we only need to compute the residuals at the red nodes. See [Thü02] for further implementation details.

Note that this multigrid algorithm can be shown to converge well as long as the function $a = a(x, y, z)$ — see our model problem description in the beginning of this section — varies slowly. However, the convergence of a standard multigrid scheme degrades drastically as soon as a begins to vary significantly from grid node to grid node. Hence, more involved multigrid approaches for this class of variable-coefficient problems involve operator-dependent interpolation formulas, cf. Section 3.3.5.2.

Since we have decided on the use of the Fortran77 programming language, the memory we need in order to store the matrices (i.e., the stencils) and the grid functions (i.e., the vectors of unknowns and the right-hand sides) belonging to the grid hierarchy can only be allocated statically when the execution of the code starts. Therefore, our multigrid implementation statically allocates a sufficiently large block of memory in the beginning of the execution and performs its own memory management afterwards. Much of the array index arithmetic is done manually using preprocessor macros. However, the subroutines implementing the time-consuming smoother (cf. Section 7.3.2.1) and the inter-grid transfer operators are based on the explicit use of 3D array data types in order to allow for as aggressive compiler optimizations as possible.

7.3.2 Performance Results

7.3.2.1 Analysis of the Standard Multigrid Implementation

Analysis of stall cycles. The runtime behavior of our standard multigrid implementation involving the access-oriented data layout[2] from Section 5.3.2 on Platform A[3] is summarized in Table 7.1, see also [KRTW02]. The leftmost column specifies the size of the finest level of the corresponding

[2]This choice is justified by the results we will present in Section 7.3.2.2 below.

[3]Compaq Tru64 UNIX V5.0A, Compaq Fortran77 V5.1, flags: `-extend_source -fast -O4 -pipeline -tune host -arch host -g3`

grid hierarchy. For all problem sizes, the performance is far away from the theoretically available peak performance of 1 GFLOPS. In particular, the performance for the two largest data sets is significantly lower than the performance of the multigrid code using smaller grids.

Although the DEC Alpha architecture found in Platform A is outperformed by all current architectures and must be considered outdated, its relatively large L3 cache of 4 MB represents current trends concerning the increase of cache capacities, both on-chip and off-chip, cf. Section 2.2.2. Moreover, the Alpha-specific profiling infrastructure DCPI enables an intimate analysis of CPU stall cycles, see [ABD+97] as well as Section 2.5.1.1.

Therefore, in order to determine the causes for the performance drops, we have profiled the program using the DCPI tool suite. The result of the analysis is a breakdown of CPU cycles spent for execution (Exec) and different kinds of stalls. Possible causes of stalls are instruction cache misses (I-Cache), data cache misses (D-Cache), data TLB misses (TLB), branch mispredictions (Branch), and register dependences (Depend); see [Bäh02a, HP03], for example. We have included the corresponding performance results into Table 7.1.

The first observation is that, for all grid sizes, instruction cache misses and incorrect predictions of conditional branches do not represent a significant performance bottleneck. For the smaller grids, instead, register dependences are the limiting factor. However, with growing grid size, the memory behavior of the code dominates its execution time. Thus, for the two largest grid sizes, data cache miss stalls account for more than 60% of all CPU cycles. In contrast to the 2D case, where data TLB misses do not play an important role, they must be taken into consideration in 3D as well. See also Sections 7.3.2.4 and 7.5.2 below.

These results clearly illustrate the poor memory performance of standard implementations of iterative algorithms. Hence, it is obvious that they motivate our research on locality optimizations, in particular our research on techniques to enhance the cache performance of this class of methods.

It is important to mention that all MFLOPS rates we present in this chapter have been obtained by explicitly counting the floating-point operations in the source codes and measuring the corresponding user times. In particular, this implies that a machine instruction which performs a vector-type floating-point operation is not counted as a single floating-point operation, but as an appropriate ensemble of floating-point operations. Examples for such vector-type machine instructions are Intel's SSE/SSE2[4] instructions, which have been added to the instruction set architectures of some Intel CPUs such as the Intel Pentium 3 (SSE only) and the Intel Pentium 4 (SSE and SSE2), for example [Int02].

[4] *Streaming SIMD Extension, Streaming SIMD Extension 2*

Grid size	MFLOPS	% of CPU cycles spent for						
		Exec	I-Cache	D-Cache	TLB	Branch	Depend	Other
9^3	67.5	28.8	0.5	14.8	4.3	1.3	39.1	11.2
17^3	55.0	20.5	1.0	37.6	9.3	0.8	21.4	9.4
33^3	56.5	19.2	1.2	41.9	15.6	0.4	15.0	6.7
65^3	22.6	9.3	1.1	63.7	18.4	0.1	4.9	2.5
129^3	20.7	9.1	1.3	60.3	20.9	0.1	3.7	4.6

Table 7.1: Runtime behavior of a standard 3D multigrid code.

Multigrid component	CPU time
Smoother (finest grid)	73.0%
Residual restriction	16.4%
Smoother (all coarser grids)	5.9%
Interpolation/correction	3.5%
Others (e.g., initialization)	1.2%

Table 7.2: Breakdown of CPU time for multigrid V(2,2)-cycles on Platform A, measured with DCPI.

Costs of the multigrid components. Table 7.2 shows another breakdown of CPU time for standard multigrid V(2,2)-cycles involving our access-oriented data layout on Platform A. In this experiment, we have measured the fractions of execution time spent in the different components of the multigrid method. Similar ratios have been obtained for other architectures. We have again chosen a standard red-black Gauss-Seidel smoother on each level of the grid hierarchy. The finest grid level contains 129^3 nodes. Once more, we have used the profiling tool suite DCPI [ABD+97], see also [Thü02].

The numbers in Table 7.2 reveal that most of the execution time is spent for smoothing the algebraic errors on the finest grid of the hierarchy, while the smoothers on all coarser grids only account for about 6% of the total user time. Furthermore, since we consider the common case of V(2,2)-cycles and since the computational effort to compute the current residual is comparable to the work needed to perform a single iteration of the smoother, the residual restriction takes about 25% of the smoothers. Here, the gain achieved by computing the residuals only at the red nodes of the corresponding fine grid (see Section 7.3.1) seems to be compensated by the work needed to sum up the residuals when setting up the right-hand side on the next coarser grid.

According to the performance data provided in Table 7.2, the most promising optimization target is the smoother on the finest grid level of the multigrid hierarchy. These observations parallel our observations for the 2D case (see [Pfä01] and [Wei01]) and motivate our optimization efforts.

It has further been shown that, in the 2D case, the additional optimization of the subroutines implementing the inter-grid transfer operators is not worth the effort at all and may even reduce the performance of the code due to its increased complexity. This additional optimization covers the fusion of the pre-smoothing step and the subsequent residual computation/restriction as well as, correspondingly, the fusion of the interpolation/correction step with the post-smoothing step [Wei01]. As a consequence of the performance results for the 2D case, we have omitted the analysis of this tuning step in 3D, since we expect the analogous behavior of the codes.

Upper bounds for performance and speedup. A *theoretical* upper bound for the overall speedup S, which can be achieved by tuning the smoother alone, can easily be derived from the application of *Amdahl's Law* [HP03]. If s denotes the fraction of execution time needed by the smoother, and if the execution time of the smoother itself can be reduced by a factor of $k < 1$, the overall speedup S is given by

$$S = \frac{1}{(1-s) + ks} \quad . \tag{7.1}$$

If we assume $s = 0.8$ (cf. the costs for the smoother presented in Table 7.2), we obtain $S < 5$ as an upper bound for the overall speedup S that can be expected.

An alternative approach to derive an upper bound for the performance of a code has been

proposed by Weiß in [Wei01]. The idea is based on a simplified model of the CPU — especially of its functional units — and considers the instruction mix of the code to determine the maximal MFLOPS rate to be expected. If a code is dominated by integer operations and load operations, for example, only a relatively low MFLOPS rate will be achieved, even if there are no cache misses or other stalls at all.

As an example, we again consider our standard multigrid implementation from above and analyze the occurrence of machine instructions. Again, we use Platform A. The results of this analysis are summarized in Table 7.3. For the larger grids, the majority of instructions are load operations and floating-point operations, followed by integer operations. Most RISC architectures — including the Alpha 21164 CPU found in Platform A — execute data load/store operations within their integer units, which therefore have to process both integer and load/store operations. Thus, for a data set with a finest grid containing 129^3 nodes, the integer units have to process more than 60% of all instructions.

In the case of the Alpha 21164 processor, the ratio of integer units to floating-point units is $1 : 1$. Thus, if we assume an ideal case where no stalls of any kind occur, and where one integer operation and one floating-point operation can be executed in each CPU cycle, the execution of all integer, load, and store instructions will take twice as long as the execution of all floating-point instructions. Consequently, this limits the achievable floating-point performance to 50% of the theoretically available peak performance.

Although this observation is idealized, it allows the derivation of an upper bound for the performance of our multigrid codes. Higher performance can only be achieved if the ratio of floating-point instructions to load/store instructions changes. The data locality optimizations we have presented in Chapters 5 and 6 preserve all data dependences and, in general, do not reduce the number of load/store instructions. Only as a side effect, some of the code transformations enable the compiler to reduce the number of load/store instructions [Wei01] by allocating the CPU registers more economically. See also [KRTW02].

7.3.2.2 Comparison of Data Layouts

In Section 5.3.2, we have introduced three different data layouts for stencil-based Gauss-Seidel-type iterative schemes on regular grids; the band-wise data layout, the access-oriented data layout, and the equation-oriented data layout. We have implemented these data layouts into our red-black Gauss-Seidel method that is used as the smoother in our multigrid codes.

Figure 7.1 shows the performance in MFLOPS which results from the use of these data layouts

Grid Size	Occurrence of instructions in %				
	Integer	Float	Load	Store	Others
9^3	40.3	22.7	27.6	3.1	6.3
17^3	29.7	27.6	33.7	3.7	5.3
33^3	22.1	31.2	38.1	4.3	4.3
65^3	17.7	32.8	40.6	4.7	4.1
129^3	15.5	33.2	40.2	6.7	4.4

Table 7.3: Instruction mix of our standard implementation involving multigrid V(2,2)-cycles in 3D.

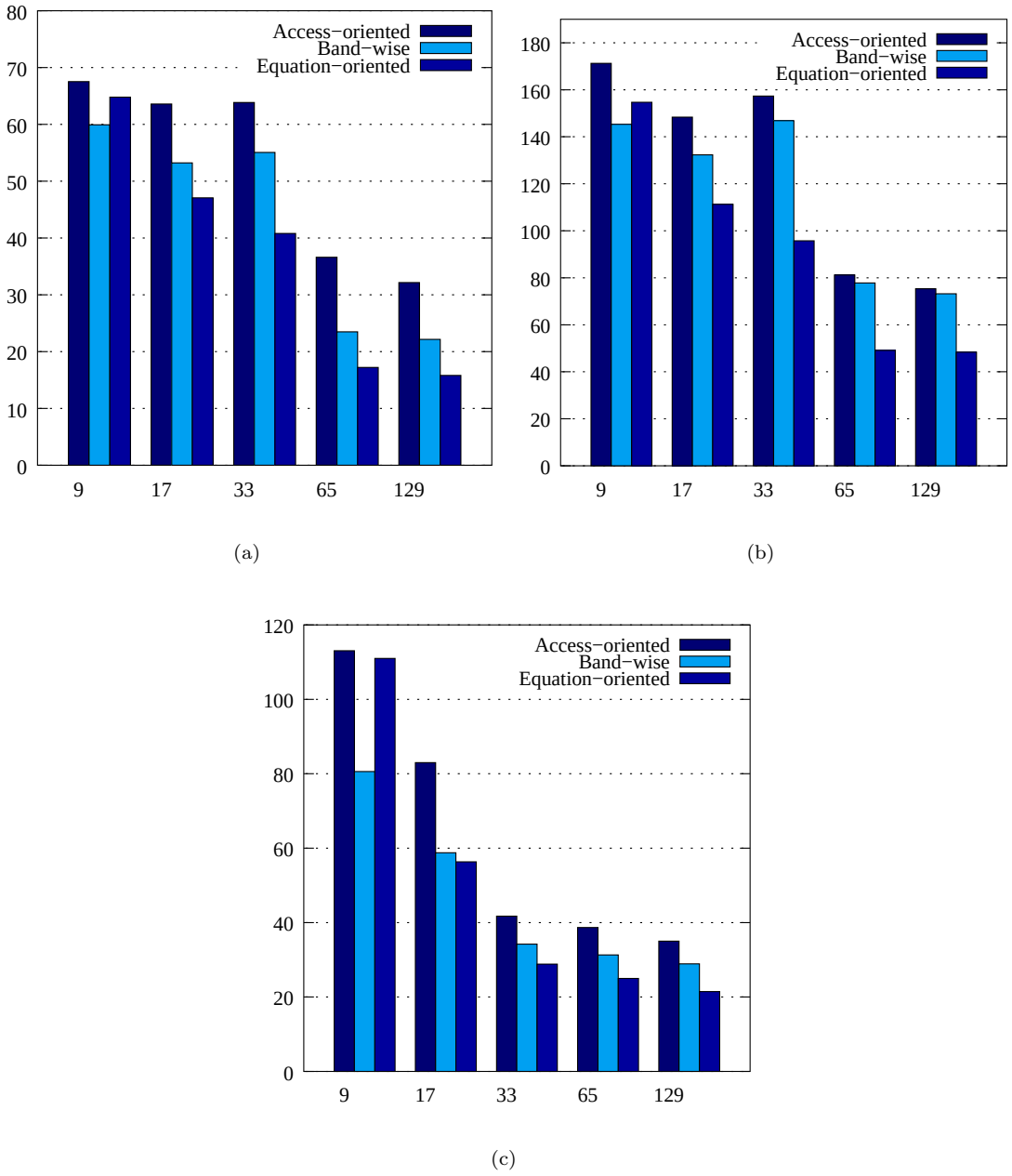


Figure 7.1: MFLOPS rates of the multigrid codes, V(2,2) cycles, for three different data layouts and varying problem sizes, (a) Platform A, (b) Platform B, (c) Platform C.

for different problem sizes on three different architectures[5]. Each of these experiments is based on the introduction of suitable array paddings in order to minimize the impact of cache conflict misses. The implementation of the multigrid method uses the most efficient loop order for the smoother, see Section 7.3.2.3 below.

Figure 7.1 reveals that the best performance is achieved as long as the access-oriented data layout is used. This corresponds to the results for the 2D case, see [Pfä01] and [KWR02]. The reason for this behavior is the good spatial locality resulting from the use of the access-oriented layout. All data merged together in memory (i.e., the entries of a matrix row and the right-hand side) is needed simultaneously when the corresponding unknown is relaxed, cf. Section 5.3.2.

On the other hand, the equation-oriented data layout yields the worst performance. The reason for this effect is the poor spatial locality of the code. Whenever an unknown is updated, the current unknowns at the neighboring grid nodes need to be accessed. In the case of the equation-oriented data layout, the unknowns are lumped together with the equation data (i.e., with the stencil weights and the right-hand sides). Therefore, for reasonably big problems that do not fit completely in cache, large portions of the cache are occupied by data which will not be accessed in the near future. This results in poor utilization of the cache.

Finally, we have observed that the performance resulting from the introduction of the band-wise data layout can vary significantly with the paddings that are introduced into the arrays declarations. This is because the data items required to compute an update for an unknown in the course of a Gauss-Seidel relaxation are relatively far away from each other in address space. Thus, the performance of the code is likely to suffer from cache conflicts or even from cache thrashing effects [Pfä01]. We refer to the classification of cache misses in Section 2.4.4.

As a consequence of these results, all previous performance experiments from Section 7.3.2.1 and all further performance experiments concerning iterative linear solvers are based on the use of the access-oriented data layout.

7.3.2.3 Comparison of Loop Orders

Figure 7.2 compares the performance and the memory behavior of the standard red-black Gauss-Seidel smoother (see Algorithm 6.3 in Section 6.3.2) which results from the introduction of different loop orders on the DEC Alpha 21164 based Platform A[6]. If the loop order is ZYX, for example, this means that the z-loop is the outermost loop and the x-loop is the innermost loop, as is the case in Algorithm 6.3. For each part of Figure 7.2, the entries in the legend from bottom to top correspond to the order of the bars from left to right.

These experiments are based on a large grid of 129^3 nodes which does not fit into any level of cache. The codes employ the access-oriented data layout as well as reasonable paddings in order to reduce the number of cache conflict misses and to avoid cache thrashing. We have run ten iterations of red-black Gauss-Seidel in each test.

The results shown in Figure 7.2 are not surprising. The code with loop order ZYX performs best since it accesses both arrays (i.e., the array storing the unknowns and the array storing the equation data) with smaller strides than all other code versions. We have already mentioned in Section 5.2.1

[5] Platform A: Compaq Tru64 UNIX V4.0D, Compaq Fortran77 V5.1, flags: `-extend_source -fast -O4 -pipeline -tune host -arch host -g3`, Platform B: Compaq Tru64 UNIX V4.0F, Compaq Fortran77 V5.2, flags: `-extend_source -fast -O4 -pipeline -tune host -arch host -g3`, Platform C: Linux with Kernel 2.4.10, GNU Fortran77, gcc 2.95.3, flags: `-O3 -fforce-addr -ffixed-line-length-none`

[6] Compaq Tru64 UNIX V4.0D, Compaq Fortran77 V5.1, flags: `-extend_source -fast -O4 -pipeline -tune host -arch host -g3`

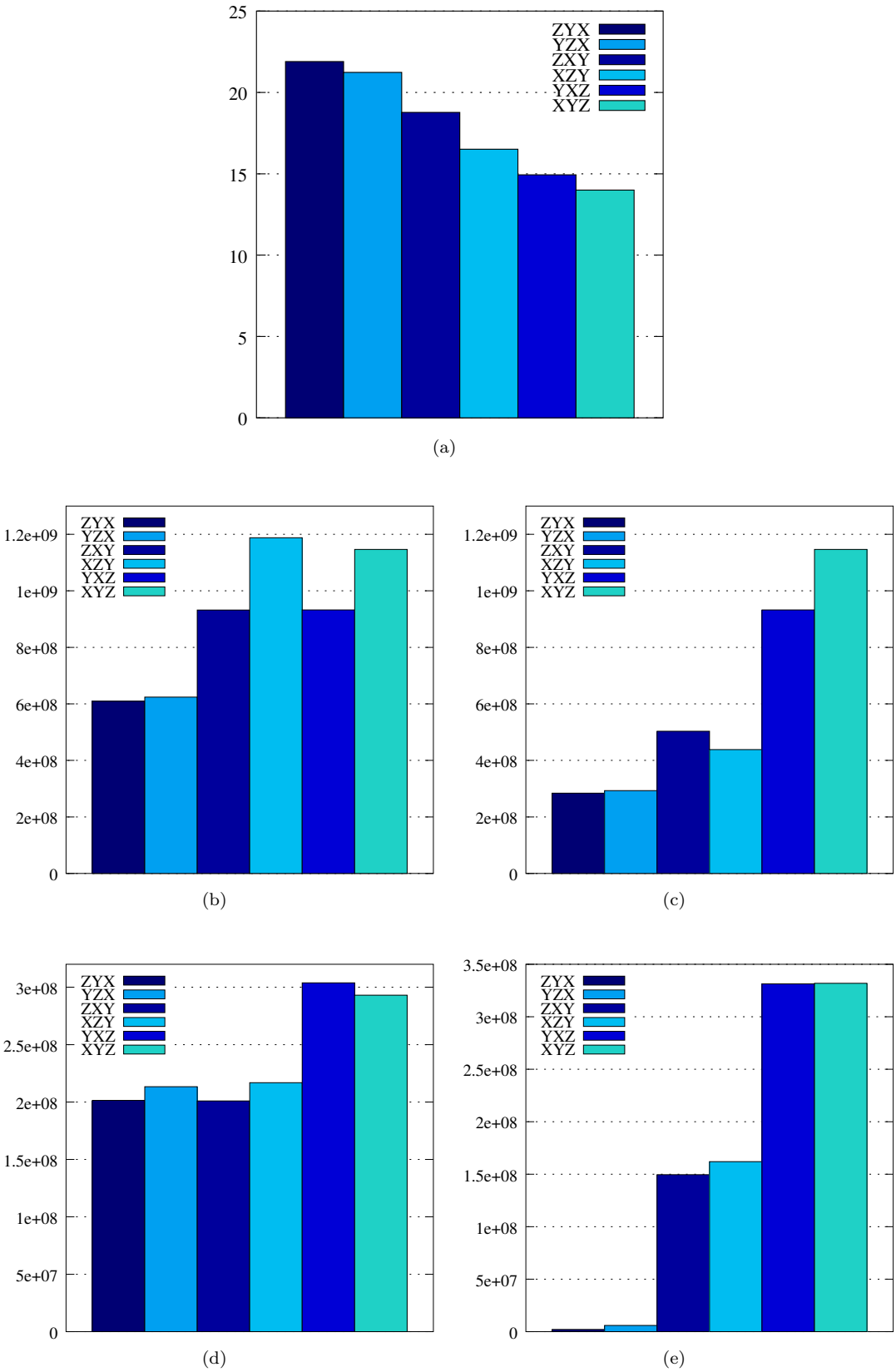


Figure 7.2: Performance results on Platform A for a standard red-black Gauss-Seidel implementation with different loop orders, (a) MFLOPS rates, (b) L1 misses, (c) L2 misses, (d) L3 misses, (e) data TLB misses.

that, in the case of Fortran77, it is the leftmost array index which runs fastest in memory, and it is the rightmost index which runs slowest. Hence, the code with loop order ZYX causes the smallest numbers of L1 misses, L2 misses, L3 misses, and data TLB misses, respectively, since it exhibits the best spatial locality among all candidates. A closer look at Figure 7.2 reveals that the order of the two outer loops hardly matters as long as the innermost loop runs along dimension x; i.e., with minimal stride 2, due to the red-black ordering of the unknowns.

In contrast, the worst performance in terms of MFLOPS results as soon as the innermost loop runs in z-direction (loop orders YXZ and XYZ). This loop order implies that the unknowns and the equation data corresponding to any two nodes that are visited in a row belong to different planes of the 3D grid. In this case, the distance in address space between two unknowns which are updated successively is twice the size of an individual plane of the grid of unknowns, again due to the imposed red-black ordering. Hence, the codes with loop orders YXZ and XYZ, respectively, only exhibit very poor spatial locality. This access pattern thus leads to poor data cache and data TLB utilization.

Note that we have included these experiments concerning the impact of the loop order for demonstration purposes only. All previous and all further experiments in this chapter are based on the use of the best possible loop orders. Of course, the inclusion of performance results for inefficient loop orders would yield more impressive speedup factors. However, we have decided to omit the presentation of artificially favorable performance results.

7.3.2.4 Multigrid Performance — General Case

The codes whose performance is analyzed in this section result from introducing the alternative smoothers into our multigrid framework, see Section 7.3.1. They employ variable coefficients and are based on the access-oriented data layout. Therefore, they represent our most general case.

For each grid node i, we need to store the seven entries $a_{i,j}$ of the i-th matrix row (i.e., the seven weights of the corresponding stencil), the corresponding right-hand side f_i, and the approximation u_i to the solution itself, see Section 3.2. Hence, each grid node occupies the space required to store nine double precision floating-point numbers. According to the international standard *IEC 559 (Binary floating-point arithmetic for microprocessor systems)*, each double precision number occupies 8 B[7] [Ueb97a]. Hence, each grid node occupies 72 B of memory.

Platform A. Figure 7.3 shows the performance in terms of MFLOPS which our codes achieve on Platform A[8]. Especially for the larger grids, the speedups resulting from the application of our cache optimizations techniques are significant. For the problems with 65^3 nodes and 129^3 nodes on the finest grid, respectively, we obtain speedups of more than a factor of 2. Note that, for all grid sizes, the code with the innermost loop being unrolled manually performs better than its 2-way, 3-way, and 4-way blocked counterparts.

Figures 7.4 and 7.5 illustrate the behavior of the respective fastest codes in detail. For this experiment, we have chosen the largest problem size with a finest grid of 129^3 nodes. Part (a) of Figure 7.4 shows that the miss rate of the rather small L1 data cache (8 kB) is hardly influenced by the application of our techniques, while the L2 data cache miss rate benefits significantly from the introduction of array padding, see Part (b). This indicates that the associativity of the L2 data

[7] Note that the standard IEC 559:1989 is also known as the *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)*, see [Gol91] for details.

[8] Compaq Tru64 UNIX V5.0A, Compaq Fortran77 V5.1, flags: `-extend_source -fast -O4 -pipeline -tune host -arch host -g3`

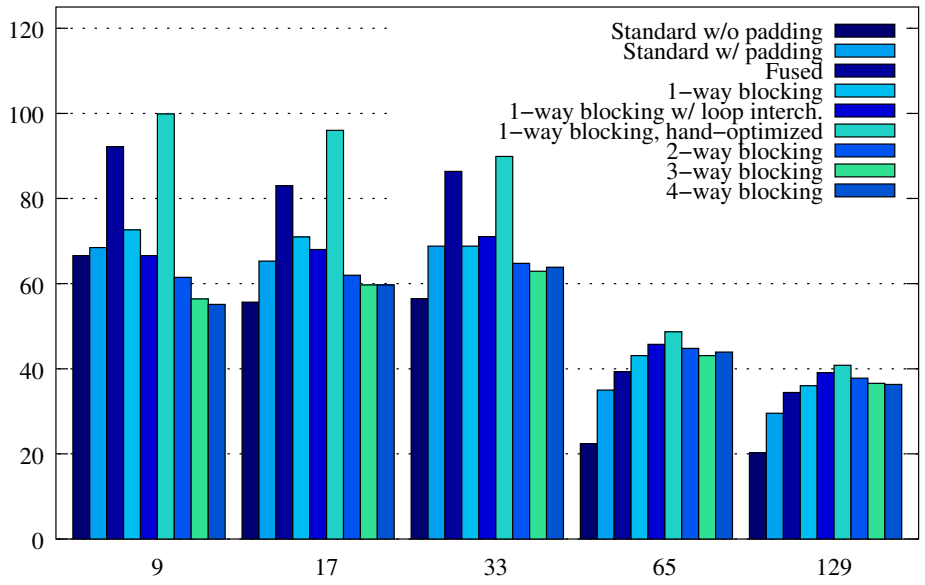


Figure 7.3: MFLOPS rates for the multigrid codes, V(2,2) cycles, with problem sizes 9^3, 17^3, 33^3, 65^3, and 129^3 using all red-black Gauss-Seidel implementations on Platform A.

cache is too low to successfully resolve mapping conflicts. Only the miss rate of the comparatively large L3 cache (4 MB) can be reduced by the successive introduction of loop fusion and 1-way blocking, see Part (c). However, more sophisticated code transformations do not further improve the L3 miss rate.

Part (d) reveals that, on the contrary, the introduction of our 4-way blocked implementation of the red-black Gauss-Seidel smoother significantly increases the data TLB miss rate. The reason for this effect is that the number of pages which are accessed in an alternating fashion is increased by the application of loop blocking, see Algorithm 6.14 in Section 6.4.3.1. Hence, the data TLB is no longer able to hold all corresponding main memory addresses. The enabling of large page support might help to circumvent this problem. See Section 7.5.2 below for the same effect in the context of the 3D LBM.

Figure 7.5 shows that the numbers of branch mispredictions and L1 instruction cache misses vary drastically, see its Parts (a) and (b), respectively. In particular, the numbers of branch mispredictions increase due to the enhanced complexities of the loop nests of the blocked implementations of the smoother. Apparently, these more complicated loop structures cause the branch prediction unit to fail more frequently. Here, manual or compiler-based loop unrolling may help to either eliminate loops with comparatively small iteration spaces or, at least, to reduce loop overhead. The relatively small number of incorrectly predicted conditional branches exhibited by the code version which is based on the manually unrolled innermost loop (*1-way blocked, hand-optimized*, see Section 7.3.1) confirms this statement, see again Part (a) of Figure 7.5.

Note that the L1 instruction cache miss rate, however, is about two orders of magnitude lower than the L1 data cache miss rate and is therefore not significant at all. Finally, Figure 7.5 reveals that the numbers of load instructions and store instructions issued by the CPU core do hardly depend on the optimization technique and need not be taken into further consideration, see Parts (c) and (d).

In summary, Figures 7.4 and 7.5 explain why the hand-optimized version of the 1-way blocked code performs best, while more sophisticated blocking approaches do not further enhance the

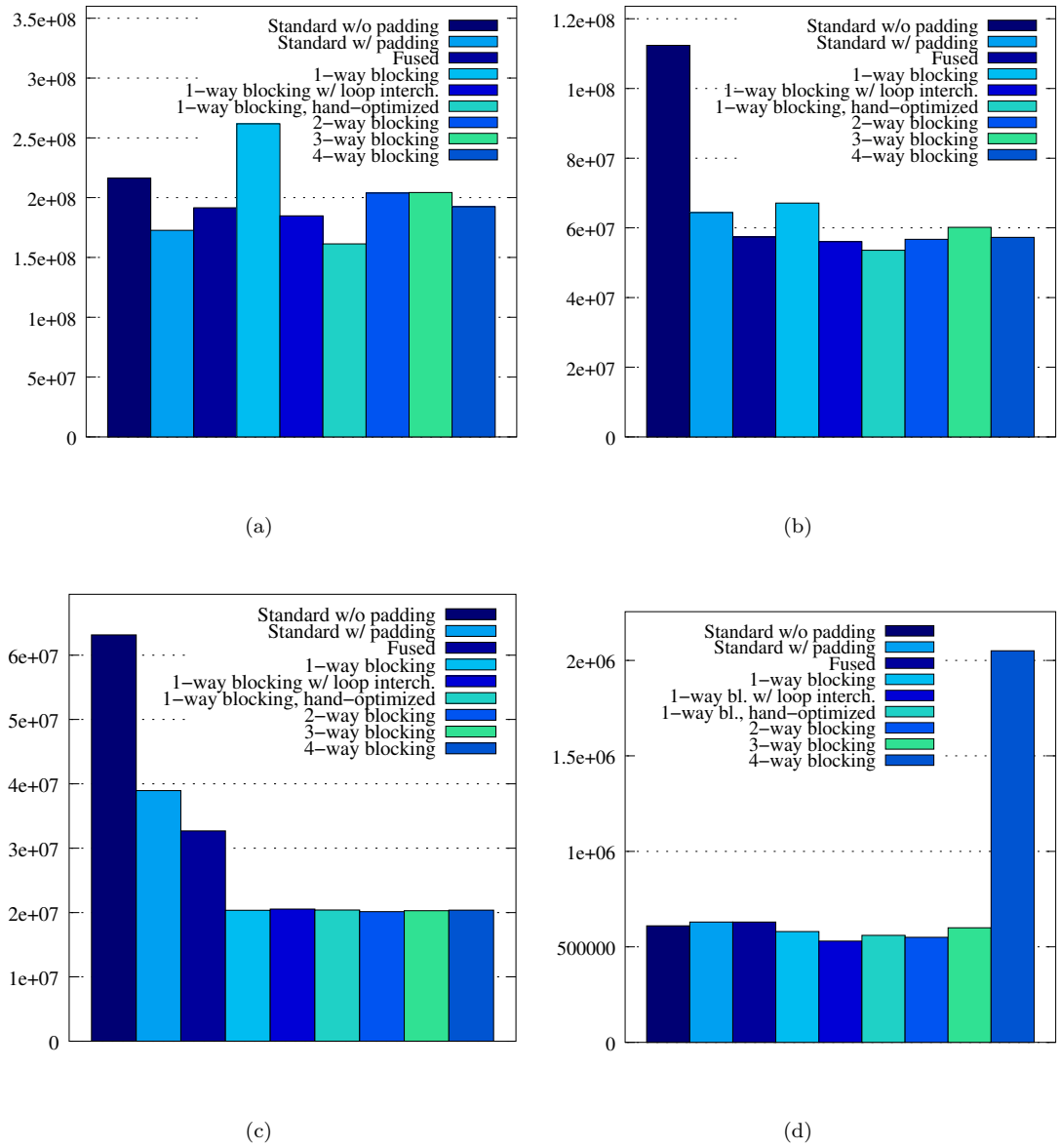


Figure 7.4: (a) L1 cache misses, (b) L2 cache misses, (c) L3 cache misses, and (d) data TLB misses on Platform A, problem size 129^3, three V(2,2) multigrid cycles, using the same paddings as in Figure 7.3.

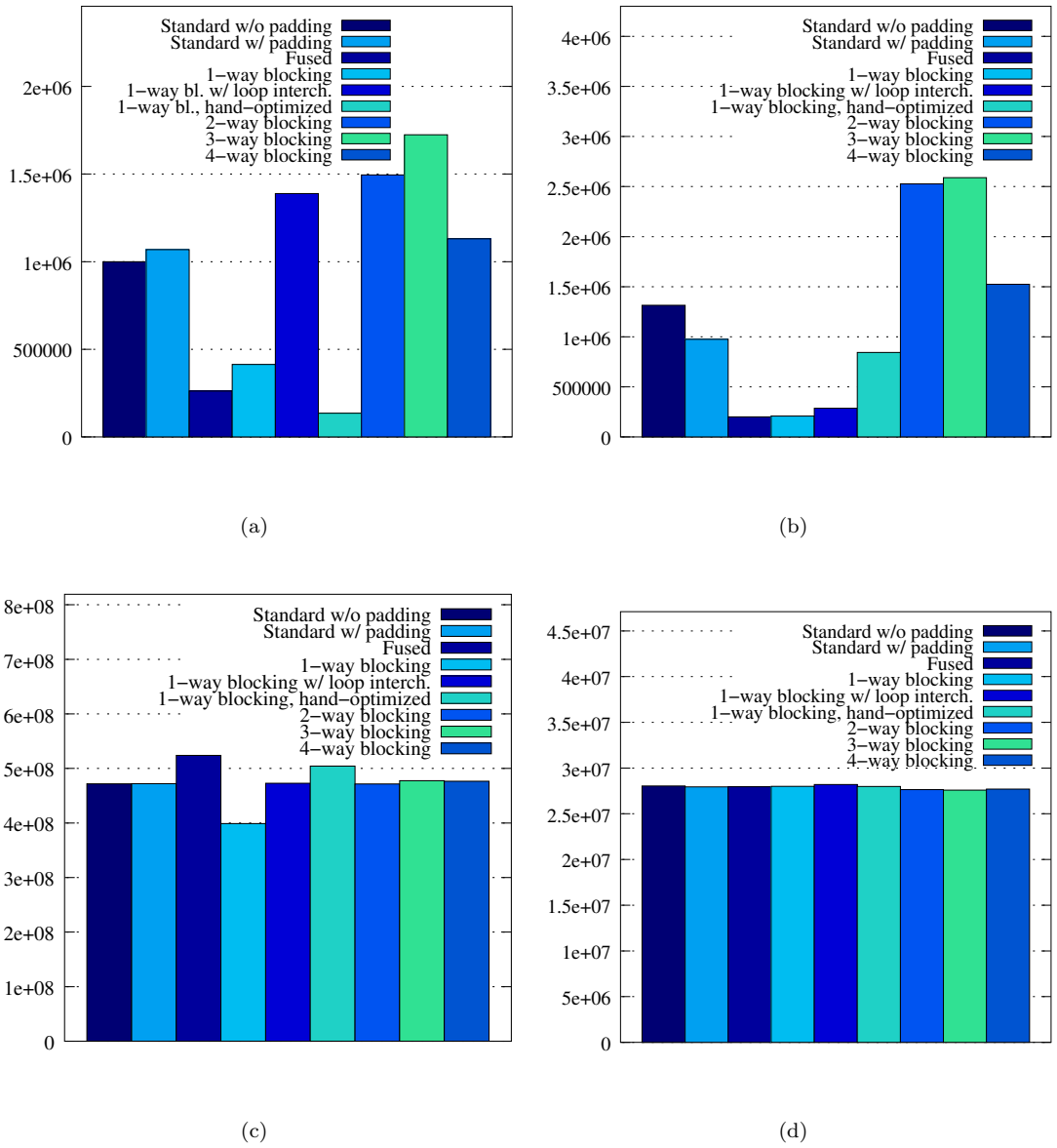


Figure 7.5: (a) Branch mispredictions, (b) L1 instruction cache misses, (c) load operations, and (d) store operations on Platform A, problem size 129^3, three V(2,2) multigrid cycles, using the same paddings as in Figure 7.3.

MFLOPS rates. In particular, it seems to be the increased number of mispredicted targets of conditional branches which is partly responsible for the performance drop resulting from the introduction of the more complicated access patterns; i.e., 2-way, 3-way, and 4-way blocking. In addition, the performance of the 4-way blocked code might further suffer from the increased data TLB miss rate. Unfortunately, these effects deteriorate the speedups that would actually result from the higher cache utilization due to loop blocking.

Figure 7.6 refers to a different experiment. In order to obtain more insight, we have investigated in how far our optimization techniques influence the individual event counts that characterize the memory behavior of the multigrid implementation. While, in the previous experiment, we have analyzed the behavior of the codes yielding the highest MFLOPS rates, the idea behind this experiment is to figure out the *minimal* numbers of L1 cache misses, L2 cache misses, L3 cache misses, and data TLB misses which result from the application of our techniques. The search for those parameters which minimize the L3 miss rate, for example, might yield a different result than the search for the parameters which maximize the performance of the code (in terms of MFLOPS).

Figure 7.6 illustrates that the L2 data cache miss rate as well as the L3 data cache miss rate can actually be reduced by the successive introduction of our optimization techniques, see Parts (b) and (c). However, Part (a) shows that the L1 data cache miss rate is again hardly influenced by the application of our approaches.

It is further noticeable that the minimal data TLB miss rate of the multigrid code involving the 4-way blocked smoother is less than half of the data TLB miss rate exhibited by the fastest multigrid code involving the 4-way blocked smoother. This results from the comparison of Figure 7.4, Part (d), with Figure 7.6, Part (d). This observation indicates that data TLB effects are only one of many factors which determine the performance (in terms of MFLOPS) of our codes and, moreover, that optimization goals may be conflicting.

As was mentioned in Section 2.5.1.1, profiling data must generally be interpreted carefully. In particular, the interaction between cache hierarchy and TLB architecture must be taken into account. Admittedly, several of the corresponding issues are not fully understood yet.

Platform D. Figure 7.7 shows the performance of our codes on the AMD Athlon XP based Platform D[9]. On this architecture, the 4-way blocked implementation of the red-black Gauss-Seidel smoother yields the highest MFLOPS rates for the two largest problems. For these data set sizes, the introduction of this access optimization technique yields a speedup of about 1.6. Recall that all code versions, except the standard implementation represented by the leftmost bars, are based

[9] Linux, GNU Fortran77, gcc 3.2.2, flags: `-O3 -fforce-addr -ffixed-line-length-none`

Hardware event	Standard	Optimized
`PAPI_L1_DCM`	$3.8 \cdot 10^6$	$2.6 \cdot 10^6$
`PAPI_L2_DCM`	$1.7 \cdot 10^6$	$1.2 \cdot 10^6$
`PAPI_TLB_DM`	$5.2 \cdot 10^4$	$8.1 \cdot 10^4$
`PAPI_L1_ICM`	72	$1.1 \cdot 10^2$
`PAPI_L2_ICM`	60	$1.1 \cdot 10^2$
`PAPI_BR_MSP`	$4.9 \cdot 10^4$	$9.0 \cdot 10^4$

Table 7.4: Profiling data measured with PAPI on Platform D.

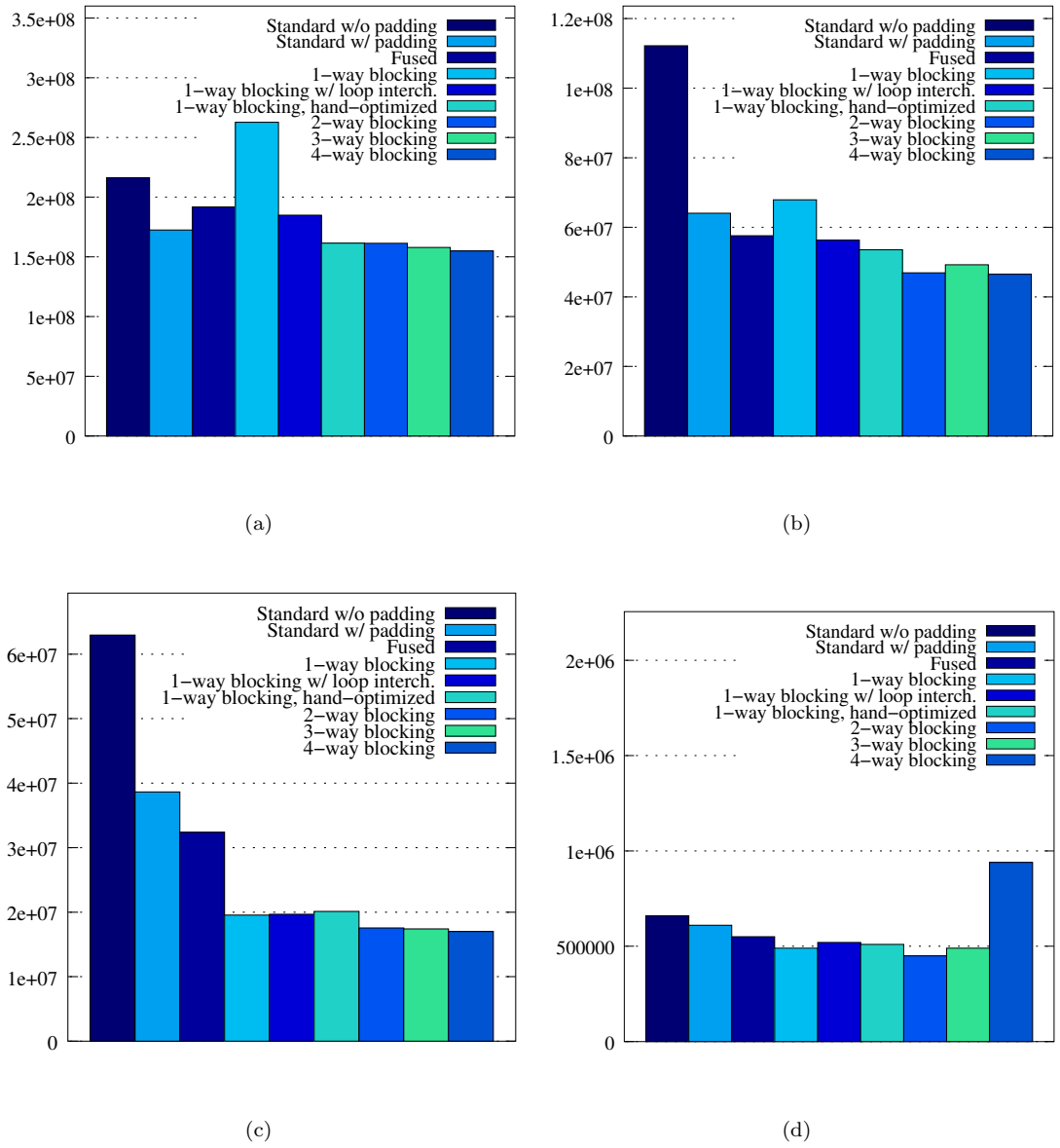


Figure 7.6: *Minimum* numbers of (a) L1 cache misses, (b) L2 cache misses, (c) L3 cache misses, and (d) data TLB misses on Platform A, problem size 129^3, three V(2,2) multigrid cycles, using suitable paddings.

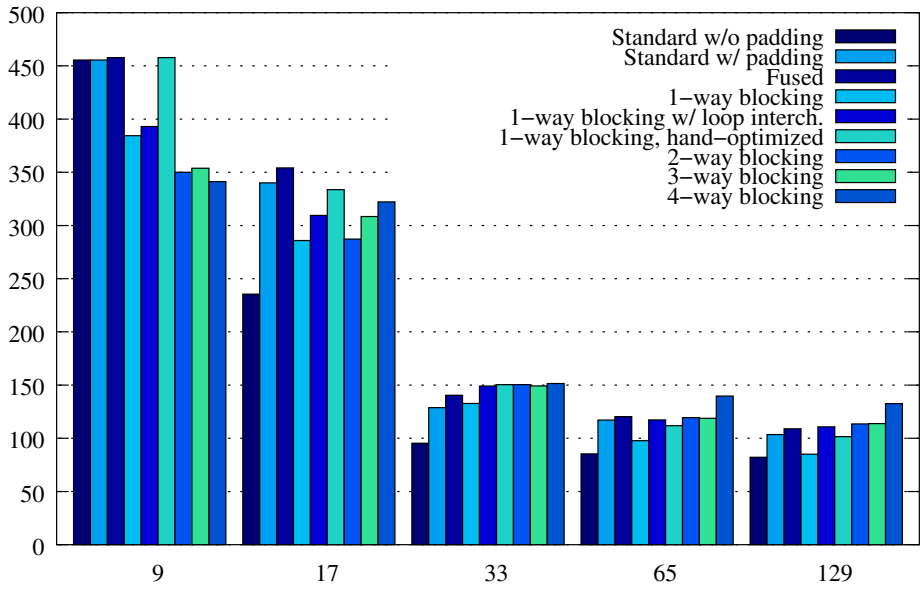


Figure 7.7: MFLOPS rates for multigrid codes, V(2,2) cycles, with problem sizes 9^3, 17^3, 33^3, 65^3, and 129^3 using all red-black Gauss-Seidel implementations on Platform D.

on the use of suitable paddings.

Table 7.4 contains performance data which has been measured on Platform D using the profiling tool PAPI [BDG+00]. The PAPI-specific names of the different hardware events have been kept. We have run a sufficiently large number of multigrid V(2,2)-cycles on a hierarchy of 3D grids with the finest grid containing 65^3 nodes. We have used a standard implementation of the red-black Gauss-Seidel smoother as well as an optimized version. The latter involves both appropriate array paddings and 4-way blocking. It corresponds to the fastest code from Figure 7.7 for this problem size. The table shows the average number of hardware events per multigrid cycle.

Apparently, the L1 as well as the L2 data cache miss rates (`PAPI_L1_DCM`, `PAPI_L2_DCM`) can be reduced by the application of suitably parameterized optimization techniques by about 30% each. These numbers are not too impressive and might be explained by the rather small cache capacities of 64 kB (L1) and 256 kB (L2), respectively. Note that, for instance, a block of 10^3 grid nodes occupies $10^3 \cdot 72 = 72,000$ B, which already exceeds the capacity of the L1 cache. This calculation results from the fact that we need to store nine double precision floating-point numbers per grid node each of which occupies 8 B, see above.

Table 7.4 further shows that the impact of instruction cache misses (`PAPI_L1_ICM`, `PAPI_L2_ICM`) is negligible. This parallels the DCPI-based measurements on Platform A, which we have presented previously.

However, the data TLB miss rate (`PAPI_TLB_DM`) increases by more than 55% since, again, the number of pages that are accessed in an alternating fashion is increased by the application of loop blocking. Therefore, the capacity of the data TLB is no longer large enough such that all corresponding main memory addresses can be stored.

Moreover, as we have already observed for Platform A, the introduction of 4-way blocking raises the number of branch mispredictions (`PAPI_BR_MSP`). On Platform D, we have obtained an increase by more than 80% due to the higher complexity of the resulting loop structure, cf. again Algorithm 6.14 in Section 6.4.3.1.

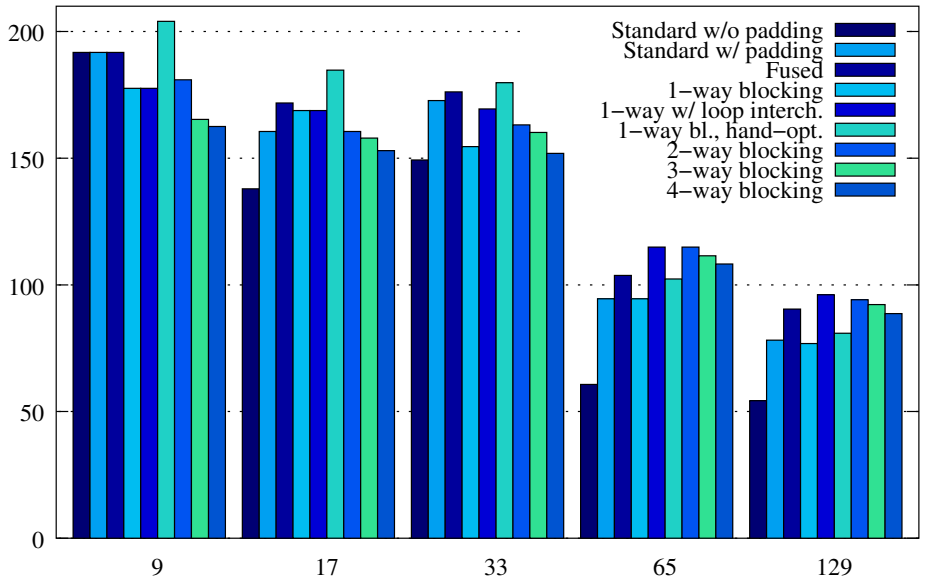


Figure 7.8: MFLOPS rates for multigrid codes, V(2,2) cycles, with problem sizes 9^3, 17^3, 33^3, 65^3, and 129^3 using all red-black Gauss-Seidel implementations on Platform B.

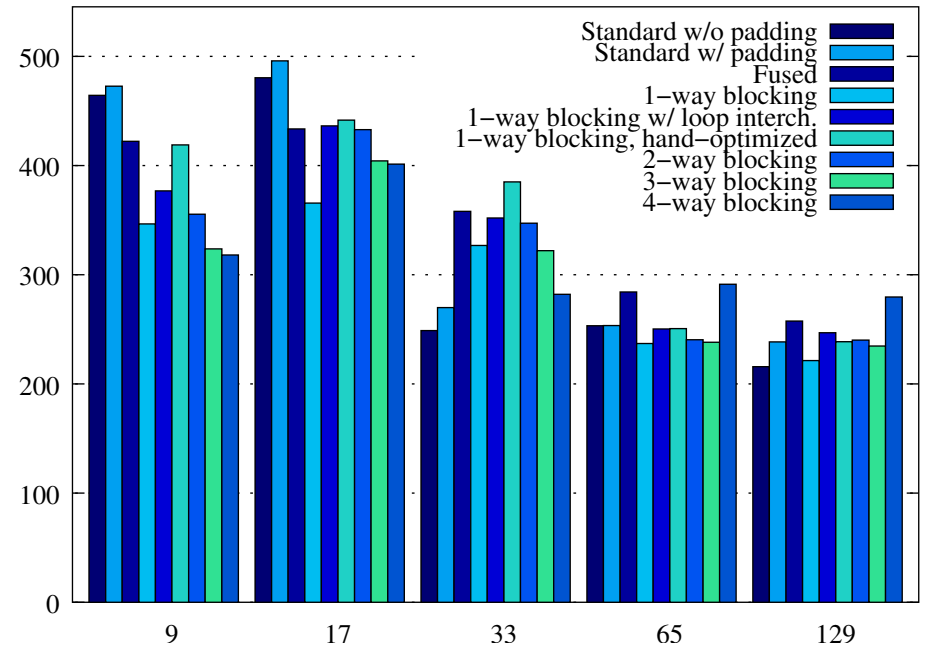


Figure 7.9: MFLOPS rates for the multigrid codes, V(2,2) cycles, with problem sizes 9^3, 17^3, 33^3, 65^3, and 129^3 using all red-black Gauss-Seidel implementations on Platform F.

Platforms B and F. Figures 7.8 and 7.9 show performance results for Platforms B[10] and F[11], respectively. Like Platform A, Platform B is characterized by a relatively large off-chip L3 cache of 4 MB. Platform B is based on a DEC Alpha 21264 CPU, which is the successor of the CPU used in Platform A. Again, for representatively large problems, speedups of about a factor of 2 can be achieved by the application of our techniques, see Figure 7.8.

However, the results obtained on the Intel Pentium 4 based machine (Platform F), which are shown in Figure 7.9, appear disappointing. For the largest data set with the finest mesh containing 129^3 nodes, for example, only the code which is based on the introduction of suitable array paddings and the 4-way blocked implementation of the smoother shows a speedup of about 30%. Obviously, this speedup is particularly far away from its theoretical upper bound, which can be derived from Amdahl's Law, cf. Section 7.3.2.1. For this problem size, our fastest multigrid code runs at about 280 MFLOPS, see the rightmost bar in Figure 7.9. Note that this MFLOPS rate corresponds to approximately 6% of the theoretically available peak performance of 4.8 GFLOPS.

Reasons for the poor performance gains can be conceived by considering the architecture of the Intel Pentium 4 CPU in more detail. Firstly, this processor is characterized by deep pipelines of up to 20 stages. Intel refers to this technology as *hyper-pipelining* [Int02]. As a consequence, pipeline stalls (e.g., caused by incorrectly predicted branch targets) are comparatively expensive. As was mentioned above, our blocked codes contain increasingly complicated loop structures which may cause branch prediction units to mispredict targets of conditional branches more frequently and, therefore, spoil performance gains due to improved cache utilization. Secondly, the Intel Pentium 4 architecture is characterized by sophisticated hardware prefetching mechanisms, see Section 6.5.1. Again, the introduction of complicated loop structures may confuse the hardware prefetching unit and thus deteriorate performance. Consequently, the deep pipeline and the data prefetching capabilities of the Intel Pentium 4 CPU favor the simple loop structures exhibited by the unblocked implementations.

We have examined the effectiveness of our cache optimization techniques on an Intel Pentium 4 machine, which is similar to Platform F. For this purpose, we have employed the OProfile tool, see Section 2.5.1.1. We have run our codes on the largest data set with the finest grid containing 129^3 nodes. It can be observed that the number of L2 misses per multigrid V(2,2)-cycle is actually reduced from about $1.5 \cdot 10^7$ (for the standard implementation) to about $9.2 \cdot 10^6$ (for the cache-tuned code involving suitable paddings and 4-way blocking). This accounts for an overall decrease of L2 cache misses of approximately 40%. The same behavior and similar numbers have further been observed with a simulation-based code analysis using the Valgrind framework, cf. Section 2.5.2.

Table 7.5 shows a more detailed comparison of these two multigrid implementations. It particularly illustrates that the number of L2 cache misses occurring during the execution of the Gauss-Seidel smoother can approximately be halved by applying both array padding and 4-way blocking, which, by itself, must be considered a positive result. Of course, the L2 cache behavior of the other multigrid components remains unchanged.

However, the OProfile-based performance analysis further reveals that the rate of incorrectly predicted branch targets increases by about 45% at the same time, deteriorating the overall speedup. This result confirms our previous considerations and emphasizes once more that optimization goals can be conflicting.

[10] Digital UNIX V4.0F, Compaq Fortran77 V5.2, flags: `-extend_source -fast -O4 -pipeline -tune host -arch host -g3`

[11] Linux, Intel ifc V7.0, flags: `-O3 -extend_source -tpp7 -xW -w`

Multigrid component	Standard	Optimized
Smoother	$1.3 \cdot 10^7$	$6.7 \cdot 10^6$
Residual calculation/restriction	$2.2 \cdot 10^6$	$2.2 \cdot 10^6$
Interpolation/correction	$1.9 \cdot 10^5$	$1.9 \cdot 10^5$
Initialization	$1.0 \cdot 10^5$	$1.0 \cdot 10^5$
Total	$1.5 \cdot 10^7$	$9.2 \cdot 10^6$

Table 7.5: Breakdown of L2 cache misses per multigrid V(2,2)-cycle on an Intel Pentium 4 machine similar to Platform F using OProfile, problem size 129^3.

7.3.2.5 Multigrid Performance — Special Case: Constant Coefficients

Figure 7.10 shows performance data for our multigrid implementation using 7-point stencils with constant coefficients on Platform A[12]. This means that we only need to store two arrays per grid level; the corresponding unknown vector as well as the corresponding right-hand side. Since, on each level of the grid hierarchy, the coefficients of the stencils (i.e., the entries of the matrix rows) do not vary from grid node to grid node, they can either be kept in registers or, at least, be retrieved quickly from the L1 cache.

The comparison of Figures 7.3 and 7.10 reveals that the standard implementation of the code involving constant coefficients runs about twice as fast as its counterpart involving variable coefficients. However, the application of our cache optimization techniques only yields a speedup of

[12]Compaq Tru64 UNIX V5.0A, Compaq Fortran77 V5.1, flags: `-extend_source -fast -O4 -pipeline -tune host -arch host -g3`

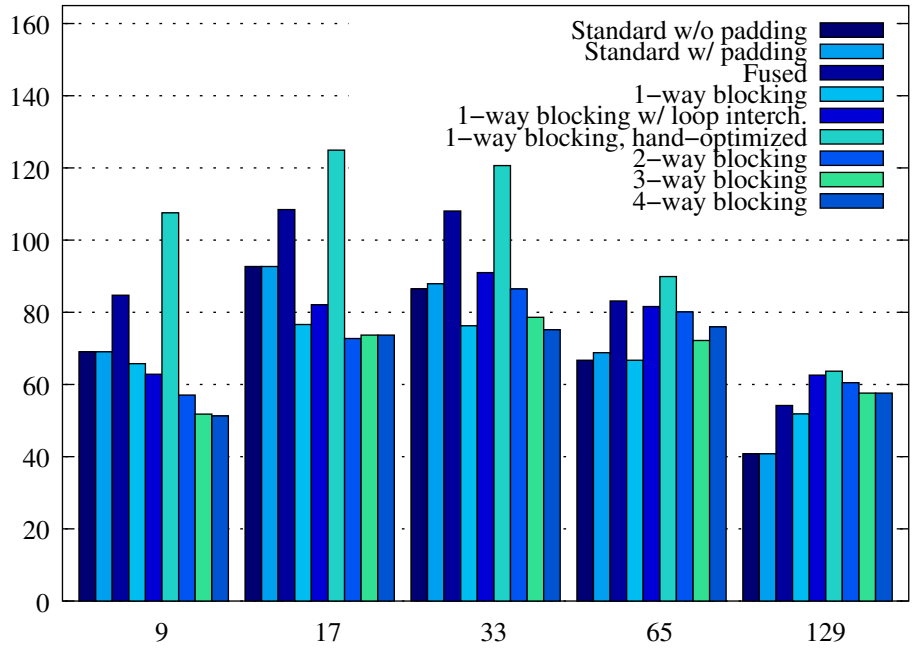


Figure 7.10: MFLOPS rates for the constant-coefficient multigrid codes, V(2,2) cycles, with problem sizes 9^3, 17^3, 33^3, 65^3, and 129^3 using all red-black Gauss-Seidel implementations on Platform A with suitable paddings.

about 50% for the largest problem size, while the speedups for the smaller problems are even lower. Again, this can be considered a negative result since, for the largest data set, the fastest multigrid code still runs at less than 7% of the theoretically available peak performance of 1 GFLOPS of Platform A.

In the 2D case, the application of suitable cache optimization techniques to the implementations of red-black Gauss-Seidel smoothers based on 5-point stencils with constant coefficients has been demonstrated to yield more impressive results. For larger problems, we have measured speedups for the whole multigrid code of about a factor of 3 on Platform A, with the fastest code running at approximately 15% of the theoretically available peak performance. See [Wei01] for detailed presentations. For the case of variable coefficients in 2D, we refer to [Pfä01].

7.4 The LBM in 2D

7.4.1 Description of the Implementations

The 2D LBM codes discussed in this section have been implemented in ANSI C++. Much attention has been paid in order to avoid common performance bottlenecks that may come along with certain C++ features such as polymorphism, for example [BM99]. Again, floating-point numbers are generally stored as double precision values. We use a standard model problem in order to examine the performance gains which can be achieved by introducing the cache optimization techniques from Chapters 5 and 6; the lid-driven cavity, see Section 4.2.7.

Note that our codes are general enough to handle more complicated scenarios; e.g., involving solid obstacles located inside the lattice. Consequently, the bodies of the innermost loops (i.e., the subroutines *update()* and *updateBlock()* used in the pseudo-codes in Section 6.4.2.2) contain the evaluation of a condition in order to determine if the current cell is an acceleration cell, a fluid cell, or an obstacle cell.

Figure 7.11 illustrates the data structure we employ to store a lattice of grid cells. White cells denote fluid cells, the cells marked with arrows represent acceleration cells and correspond to the lid which is permanently dragged across the fluid. The fluid cells and the acceleration cells are surrounded by a layer of obstacle cells (shaded gray). In order to simplify the implementation, we have added an additional layer of fluid cells around this layer of obstacle cells.

In each time step, all cells are updated, except those belonging to the outermost layer of fluid cells. The purpose of this outermost layer of fluid cells is to enable all interior cells to be updated using the same access pattern; i.e., by accessing eight neighboring cells. Hence, no special handling is required for the update of the layer of obstacle cells. Note that, due to the implementation of the obstacle cells using the bounce-back assumption from Section 4.2.5, no information from outside the closed box can influence the behavior inside, and vice versa. Figure 7.11 further reveals that the additional layer of fluid cells is not counted as far as the problem size is concerned. In other words, in a grid of size n^2, there are n^2 interior cells that are actually updated in each time step.

If the data layout based on two separate grids is used, two separate data structures such as the one shown in Figure 7.11 need to be allocated in memory. In the case of array merging, each cell in Figure 7.11 actually represents a pair of lattice sites; one component corresponding to the source grid and the other component corresponding to the destination grid, see Section 5.3.3.

Finally, if the grid compression scheme is employed, the data structure that needs to be allocated in memory contains further rows and columns of cells, depending on the number t_B of time steps to be blocked into each single pass through the grid. Recall Sections 5.4 and 6.4.2.2 for details.

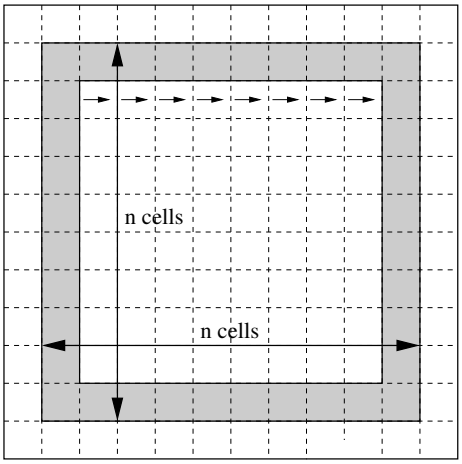


Figure 7.11: Implementation of the 2D LBM.

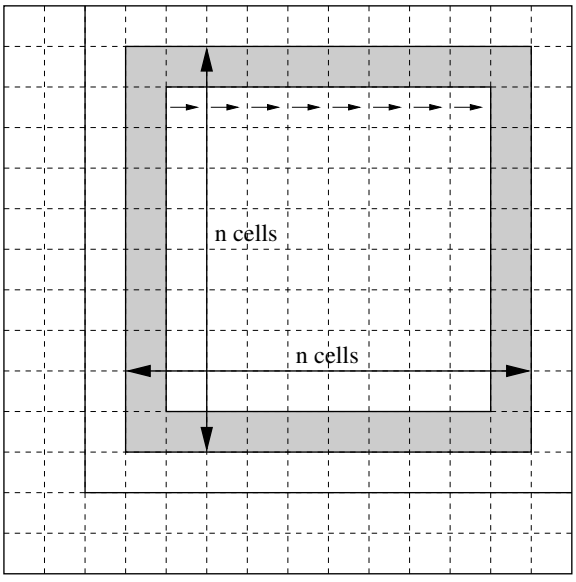


Figure 7.12: Implementation of the 2D LBM using the grid compression data layout with two consecutive time steps being blocked.

Figure 7.12 illustrates the data layout for two consecutive time steps being blocked (i.e., $t_B = 2$), using either 1-way or 3-way blocking. Note that this figure again refers to a problem of size n^2. For a more comprehensive description of the 2D LBM implementations, we refer to [Wil03].

We have counted that, in our D2Q9 LBM model, the update of a fluid cell requires 116 floating-point operations to be executed, while the update of an obstacle cell or an acceleration cell is obviously less expensive. Consequently, if one of our 2D LBM codes runs at k MLSUPS, it performs *about* $116 \cdot k$ MFLOPS. Yet, with a growing grid size, the impact of the obstacle cells and the acceleration cells, that actually spoils this conversion factor, vanishes in the case of our lid-driven cavity setting.

7.4.2 Performance Results

MLSUPS rates. Figures 7.13 to 7.17 demonstrate the performance of five different implementations of the 2D LBM on Platforms A[13], B[14], D[15], E[16], and G[17] for various grid sizes. Note that all experimental results in this section are based on the stream-and-collide update order only.

Commonly, the standard implementation involving a merged source and destination grid is the slowest, while the implementation using the layout based on grid compression tends to be slightly faster. Two implementations of 1-way blocking are included; one with two time steps blocked (i.e., $t_B = 2$, see Algorithm 6.7 in Section 6.4.2.2), and a second one with eight time steps blocked (i.e., $t_B = 8$). Initially, they show a massive increase in performance. For large grids, however, performance degrades since the required data cannot fit into even the lowest level of cache.

This effect is particularly dramatic on the AMD Athlon XP processor (Platform D) since it has only 256 kB of L2 and no L3 cache, see Figure 7.15. Note that, if the grid compression layout is used and $t_B = 8$ successive time steps are blocked into a single pass through the grid, a single row of cells of a 1000^2 lattice occupies $(1000 + 2 + 8) \cdot 10 \cdot 8 = 80800$ B, which is already more than 30% of the size of L2 cache of the AMD Athlon XP processor.

This formula can be derived as follows. In addition to the interior cells, there are two outermost fluid cells per grid row (see Section 7.4.1). Since the grid compression scheme is employed, the blocking of the time loop requires $t_B = 8$ additional rows and columns; we refer to Section 6.4.2.2. In our implementations, each lattice site contains ten floating-point numbers (nine distribution functions and the flag indicating the type of the cell), and each double precision number occupies 8 B, cf. Section 7.3.2.4.

Note that, in contrast to Platform D, the four other architectures are characterized by comparatively large off-chip (Platforms A and B with 4 MB each) or on-chip caches (Platforms E and G with 1 MB and 3 MB, respectively). We refer to the specifications in Appendix A for technical details.

On each architecture under consideration, the 3-way blocked implementation shows a significant increase in performance for all grid sizes. Moreover, except for isolated effects based on cache thrashing, the MLSUPS rates of the 3-way blocked codes are independent of the grid size, which corresponds to our expectations from Section 6.4.2.2. We have chosen $x_B = y_B = t_B = 16$ in all experiments of this section, see Algorithm 6.8 in Section 6.4.2.2.

It should further be noted that, in these experiments, each blocked implementation based on

[13] Compaq Tru64 UNIX V5.0A, Compaq cxx V6.5, flags: `-O4 -arch host -tune host -fast -g3`
[14] Compaq Tru64 UNIX V4.0F, Compaq cxx V6.5, flags: `-O4 -arch host -tune host -fast -g3`
[15] Linux, GNU gcc 3.2.1, flags: `-O3 -ffast-math`
[16] SuSE Linux SLES 8 (AMD64), Kernel 2.4.21 (SMP), GNU gcc 3.3, flags: `-O3 -ffast-math`
[17] Linux, Intel ecc V7.0, flags: `-O3 -ip -tpp2`

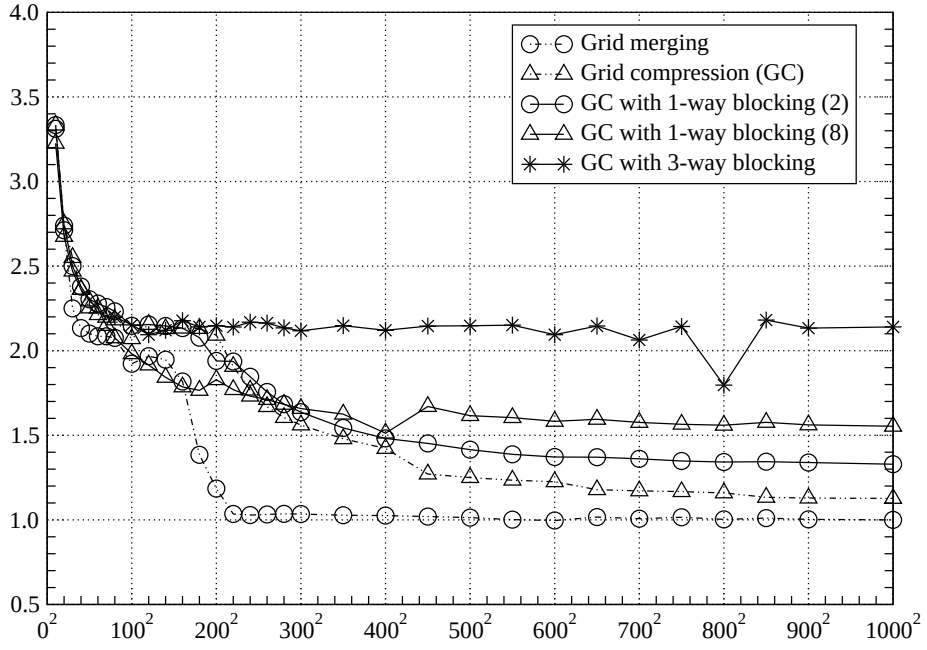


Figure 7.13: MLSUPS rates for various 2D LBM codes and problem sizes on Platform A.

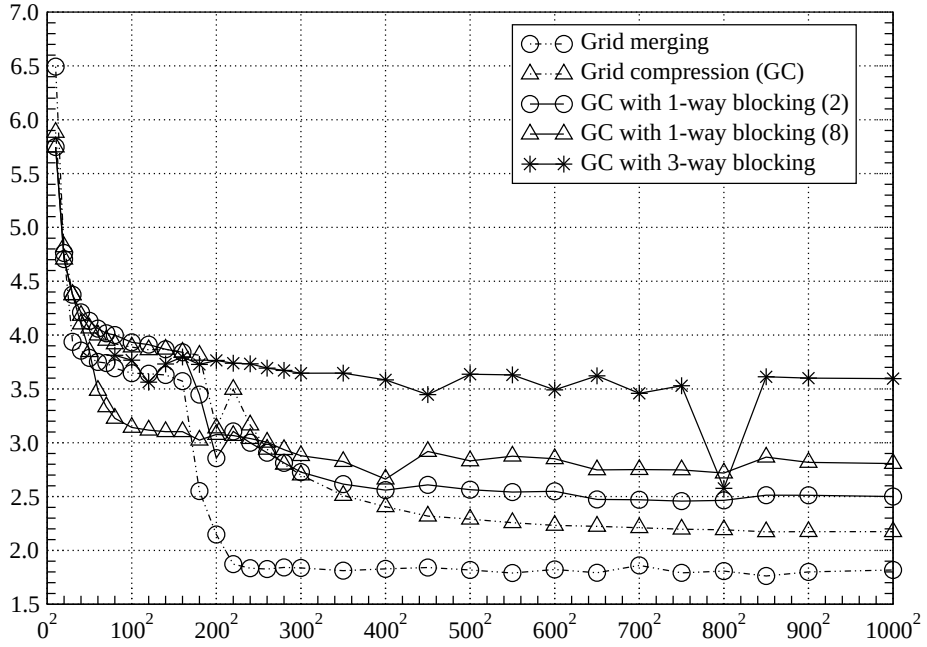


Figure 7.14: MLSUPS rates for various 2D LBM codes and problem sizes on Platform B.

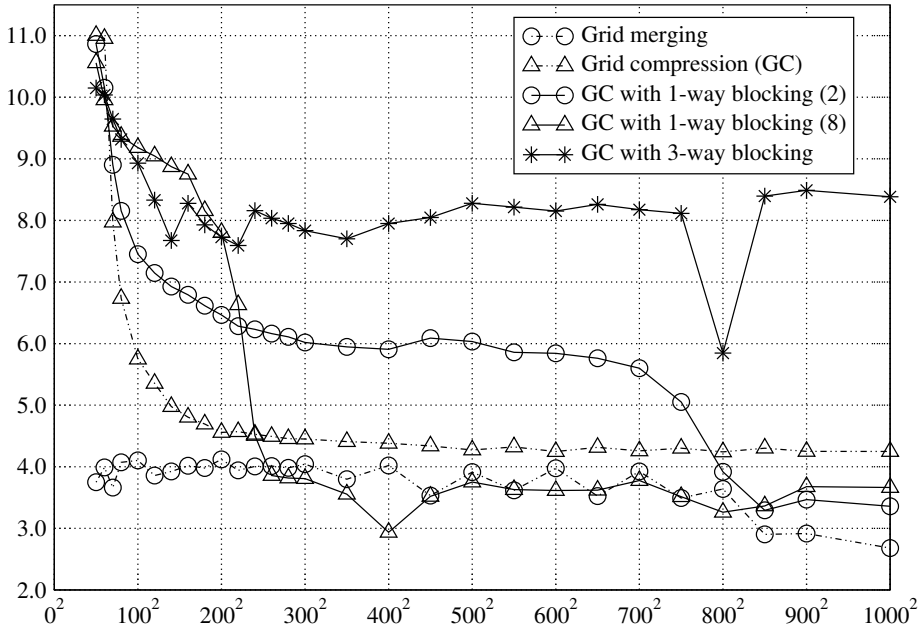


Figure 7.15: MLSUPS rates for various 2D LBM codes and problem sizes on Platform D.

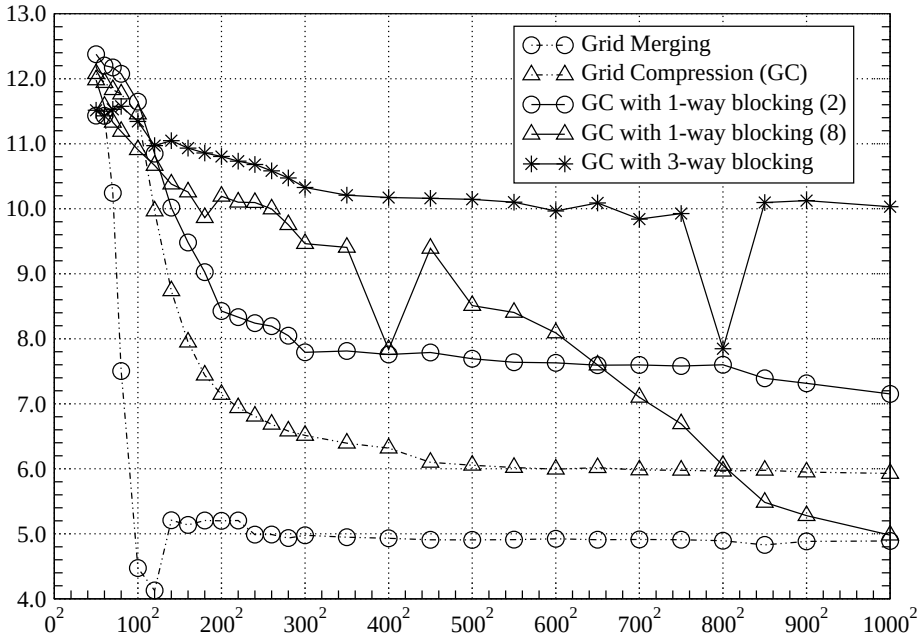


Figure 7.16: MLSUPS rates for various 2D LBM codes and problem sizes on Platform E.

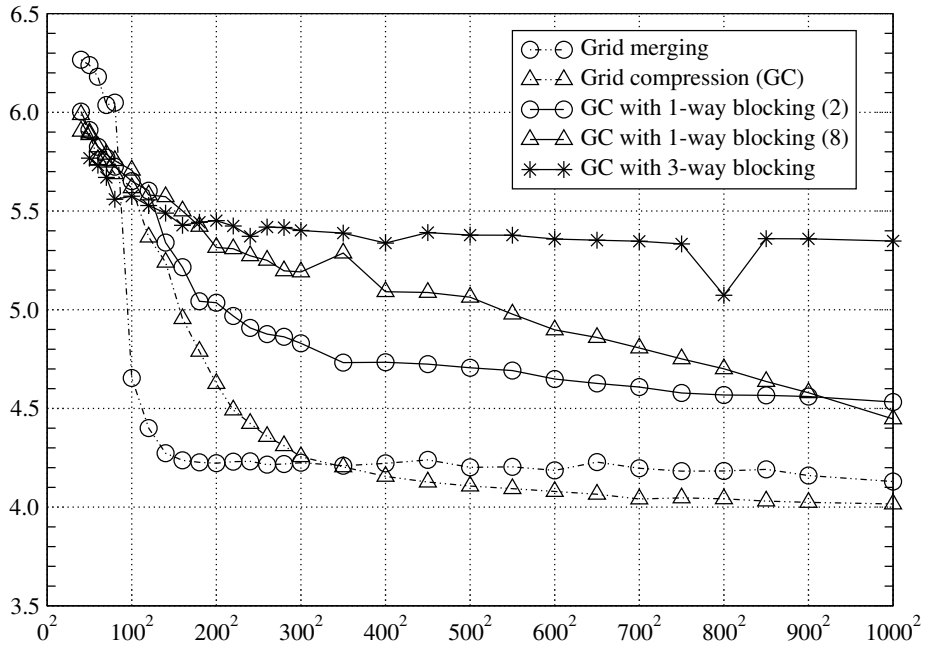


Figure 7.17: MLSUPS rates for various 2D LBM codes and problem sizes on Platform G.

the grid merging layout performs worse than its counterpart based on grid compression. Therefore, no performance results for blocked codes using the grid merging technique have been included into the figures.

Profiling data. Figures 7.18 and 7.19 illustrate the behavior of the L1 cache and the L2 cache of Platform D, respectively. We have normalized the profiling data by providing the average numbers of L1 and L2 misses per lattice site update, respectively. The two figures refer to the same implementations as Figure 7.15. The results have been obtained by using the profiling tool PAPI [BDG+00], see also Section 2.5.1.1. When compared to Figure 7.15, it can be observed that there is a very strong correlation between the number of cache misses and the performance of the code.

In particular, when comparing Figure 7.15 and Figure 7.19, the correlation between the performance drop of the two 1-way blocked implementations and their respective rise in L2 cache misses becomes especially apparent. The performance of the 1-way blocked version with $t_B = 8$ drops (and the corresponding L2 miss rate rises) as soon as the grid size grows larger than 200^2. In contrast, the MLSUPS rate resulting from the 1-way blocked code with $t_B = 2$ only decreases drastically (and, again, the corresponding L2 miss rate increases drastically) as soon as the problem size becomes larger than 700^2.

Additionally, Figure 7.18 reveals the cause of the severe performance drop exhibited by the 3-way blocked implementation at a grid size of 800^2. The high number of L1 misses are conflict misses which occur due to the limited set associativity of the cache, see Section 2.4.4. Note that the L1 cache of the AMD Athlon XP processor is only 2-way set-associative, see Appendix A for technical details.

This cache thrashing effect can be explained as follows. If the grid compression technique is used and $t_B = 16$ time steps are blocked into a single pass through the grid, $(800 + 2 + 16)^2 = 818^2$ cells must be allocated in memory, cf. again Section 6.4.2.2. Hence, a single row of the grid contains $818 \cdot 10$ floating-point numbers. Recall that the nine distribution functions of each cell in the D2Q9

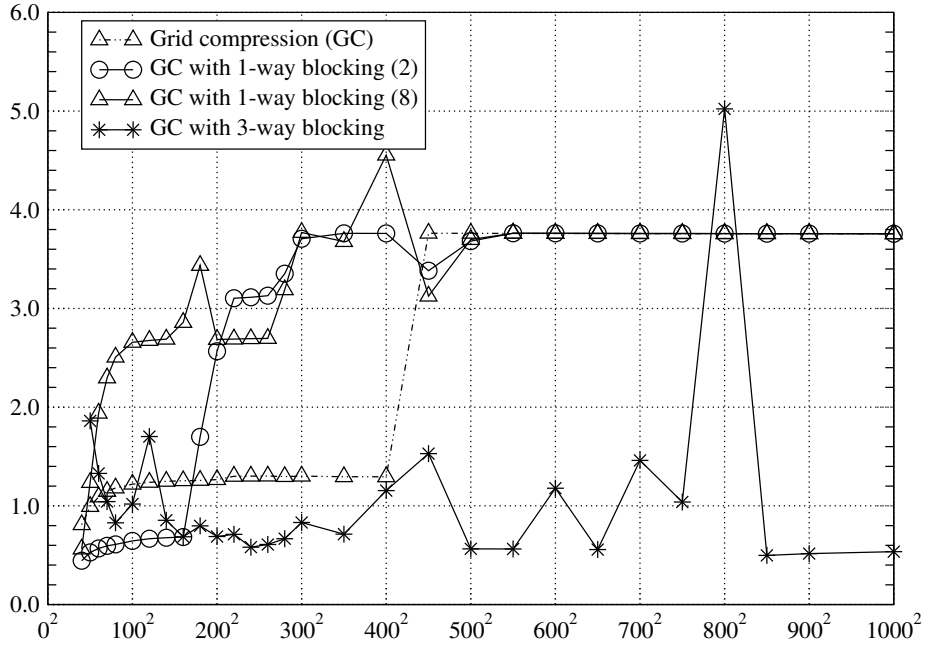


Figure 7.18: Average number of L1 cache misses per lattice site update for various 2D LBM codes and problem sizes on Platform D.

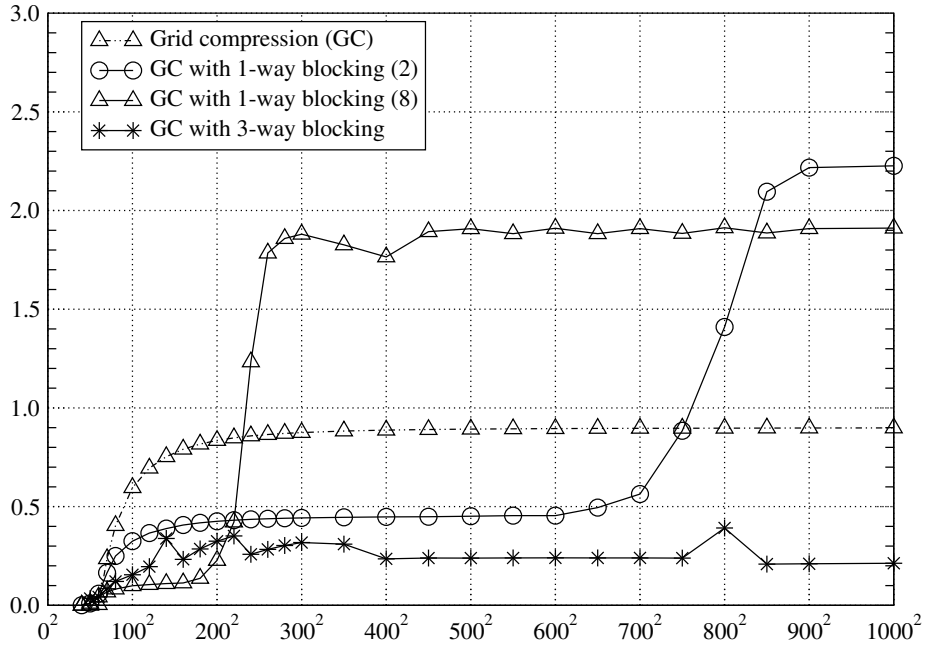


Figure 7.19: Average number of L2 cache misses per lattice site update for various 2D LBM codes and problem sizes on Platform D.

lattice as well as the flag indicating the type of the cell are stored as an array of ten floating-point numbers. Since we use double precision values each of which occupies 8 B, a single row of the grid is $818 \cdot 10 \cdot 8\text{B} \approx 64\text{kB}$ large, which equals the size of the L1 cache of the AMD Athlon XP CPU. Hence, accesses to neighboring cells which belong to adjacent rows of the grid exhibit a high potential for cache conflict misses.

A variety of further performance experiments concerning cache-optimized LBM codes in 2D can be found in [Wil03]. We also refer to our publications [WPKR03a, WPKR03b] as well as the corresponding extended versions [PKW+03a, PKW+03b].

7.5 The LBM in 3D

7.5.1 Description of the Implementations

Our 3D LBM codes have been implemented in ANSI C. As is the case for our 2D LBM codes, all floating-point numbers are represented as double precision values. Again, we use the lid-driven cavity as our model problem, see Section 4.2.7. Like our 2D LBM codes, our 3D codes are characterized by the evaluation of a condition inside the body of the innermost loop such that obstacles which are located in the interior of the box can be handled as well.

The grid data structures that we have introduced are canonical extensions of the data structures for the 2D case, see Section 7.4.1. The fluid cells as well as the acceleration cells representing the lid are surrounded by a layer of obstacle cells which in turn is surrounded by a layer of fluid cells in order to simplify the implementations. All grid sites except those belonging to this outermost layer of fluid cells are updated in each time step. As was explained in Sections 5.4 and 6.4.3.2, additional layers of lattice sites need to be allocated as soon as both the grid compression layout is used and loop blocking is applied to the time loop.

We refer to [Igl03] for a comprehensive description of further implementation details. In particular, we have studied alternative data access policies which aim at manually optimizing array index calculations; e.g., using sophisticated pointer arithmetic to access adjacent lattice sites. Note that these data access policies have no immediate impact on the memory hierarchy utilization of the codes and, therefore, cannot be considered as data locality optimizations. This means that they have nothing to do with the data access optimizations which we have presented in Chapter 6. The impact of these access policies on the execution speed of the codes is highly platform-dependent. A discussion of these policies is beyond the scope of this work, and we again refer to [Igl03] for a presentation of a variety of performance experiments.

In our D3Q19 LBM model, the update of a fluid cell requires 255 floating-point operations to be executed, while the update of an obstacle cell or an acceleration cell is less expensive. Consequently, a 3D LBM code (based on a D3Q19 lattice) running a k MLSUPS performs *about* $255 \cdot k$ MFLOPS. As is the case in 2D, with a growing grid size, the impact of the obstacle cells and the acceleration cells, which actually spoils this conversion factor, becomes more and more negligible for the case of our model problem; the lid-driven cavity in 3D.

7.5.2 Performance Results

MLSUPS rates. Since the update of a fluid cell in a D3Q19 LBM model involves more than twice the number of floating-point operations than a fluid cell update in a D2Q9 lattice, it is not surprising at all that the MLSUPS rates in 3D are generally lower than in 2D. However, the speedup

results for the 3D LBM are qualitatively similar.

In Figures 7.20 to 7.23, we provide performance results for Platforms A[18], D[19], E[20], and G[21]. These figures also contain performance data for the LBM codes based on our 3-way blocking approach. Recall that we have omitted a presentation of this technique in Section 6.4.3.2 and referred to [Igl03] instead.

Except for Platform D, the 4-way blocked codes generally perform significantly better than their 3-way counterparts. The reason for this effect might be that the AMD Athlon XP processor found in Platform D has only 256 kB of L2 cache and no further L3 cache. Hence, the capacity of the cache hierarchy is easily exceeded and sophisticated code transformations based on loop blocking do not lead to remarkable speedups for the case of the 3D LBM.

We have used identical block sizes on all platforms. For the 3-way blocked codes based on the grid compression layout, we have chosen $x_B = y_B = z_B = 16$. The 4-way blocked codes that employ the grid compression layout scheme further use $t_B = 4$. See again Algorithm 6.15 in Section 6.4.3.2. These sets of block sizes have proven to yield good speedups on all architectures under consideration. Certainly, there is still room for fine tuning and it is very likely that further speedups can be obtained.

In particular, the DEC Alpha 21164 (Platform A, Figure 7.20) and the AMD Opteron (Platform E, Figure 7.22) systems show promising speedups of approximately a factor of 2. The effectiveness of our cache optimization techniques on Platform A will be demonstrated below.

Results for the grid merging technique have been omitted due to the better performance of the grid compression approach. Only on the Intel Itanium 2 machine (Platform G), the 4-way blocked code employing two separate grids with $x_B = y_B = z_B = 8$ and $t_B = 4$ outperforms its counterpart based on grid compression. Therefore, we have included the corresponding performance data in Figure 7.23.

Platform G belongs to Intel's IA-64 product family and represents a so-called *EPIC (Explicitly Parallel Instruction Computing)* architecture. This means that the compiler must appropriately aggregate machine instructions to bundles. The instructions belonging to a bundle are executed in parallel by the CPU. Hence, the execution speed of a code crucially depends on the compiler to efficiently reorder and group the instructions. According to our experience, EPIC-aware programming can significantly assist current optimizing IA-64 compilers and thus improve the performance of the code. Since our codes are not particularly tailored for EPIC architectures, further speedups can definitely be achieved.

The case of a standard 3D LBM implementation using two separate grids in combination with the stream-and-collide update order is included into Figure 7.23 in order to emphasize that the performance on Platform G benefits most significantly from switching to the alternative collide-and-stream update order. Changing the order of the update step in this standard version yields MLSUPS rates that are already close to those obtained for the grid compression scheme and the collide-and-stream update pattern. The performance gap due to the two different update steps is thus extremely dramatic for this platform. However, the introduction of further optimizing code transformations does not cause improvements of more than 25%. The tuning of numerically intensive codes for this architecture must therefore be subject to future research. We refer to [Igl03] for additional details and experiments.

[18] Compaq Tru64 UNIX V5.0A, Compaq cc V6.1, flags: `-O4 -fast -tune host -g3`
[19] Linux, Kernel 2.4.21, GNU gcc 3.2.2, flags: `-Wall -Werror -O3 -march=athlon-xp -mcpu=athlon-xp`
[20] SuSE Linux SLES 8 (AMD64), Kernel 2.4.19 (NUMA), GNU gcc 3.2.2, flags: `-O3`
[21] Red Hat Linux 7.3, Kernel 2.4.20 (SMP), Intel ecc 7.1, flags: `-O3 -tpp1 -fno-alias`

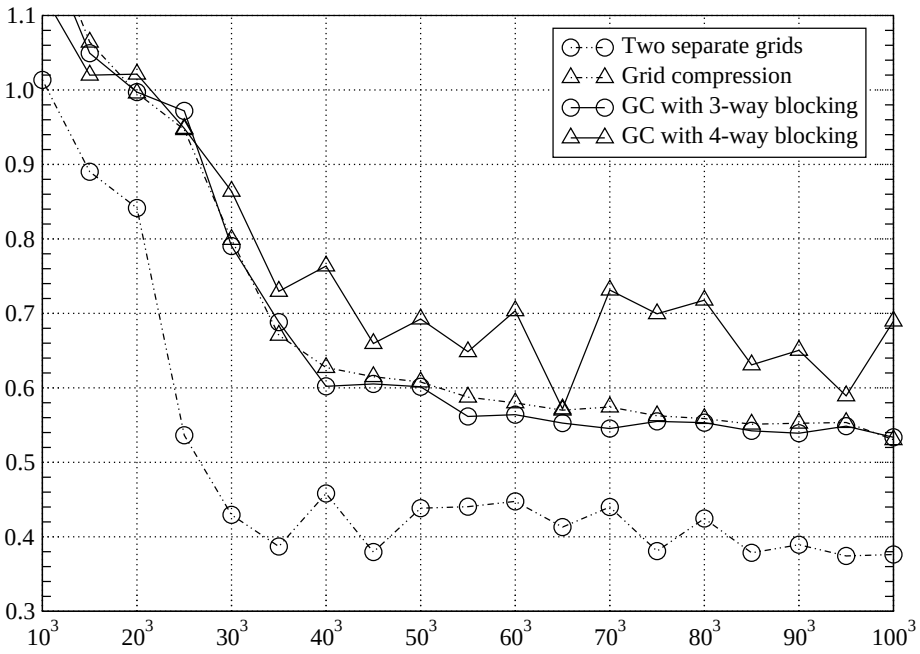


Figure 7.20: MLSUPS rates for various 3D LBM codes and problem sizes on Platform A, collide-and-stream update order only.

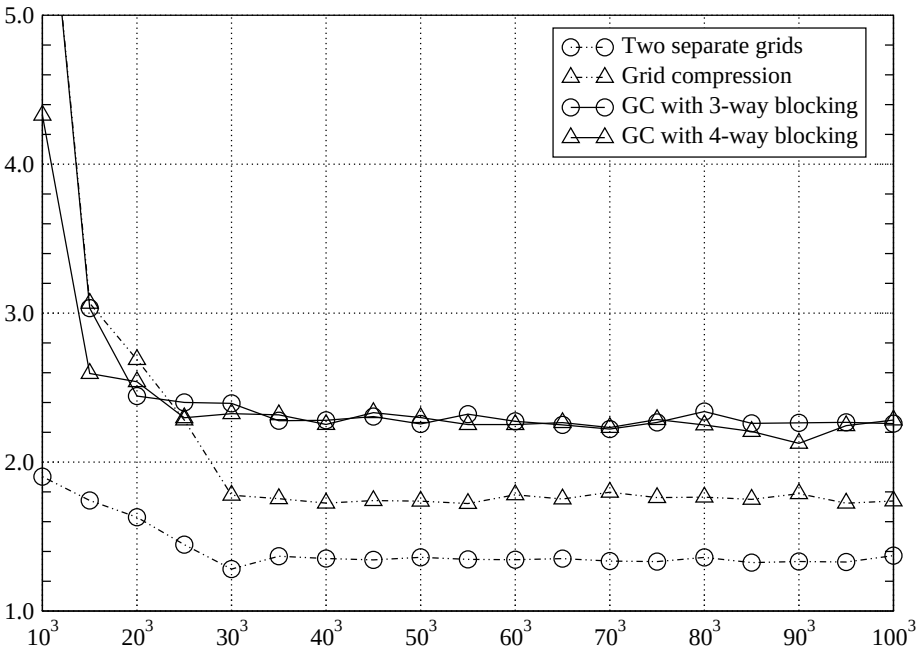


Figure 7.21: MLSUPS rates for various 3D LBM codes and problem sizes on Platform D, stream-and-collide update order only.

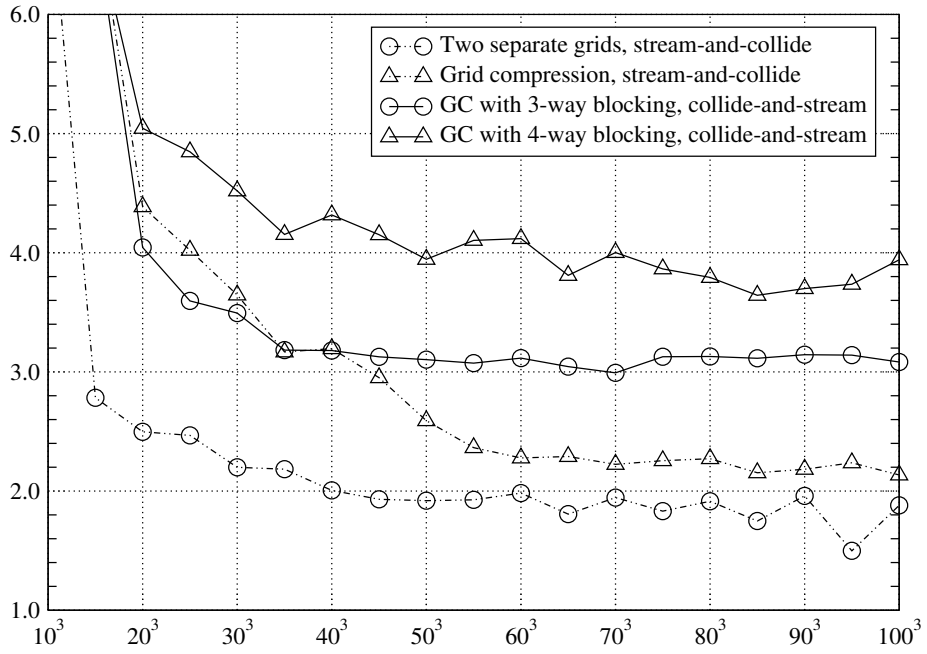


Figure 7.22: MLSUPS rates for various 3D LBM codes and problem sizes on Platform E.

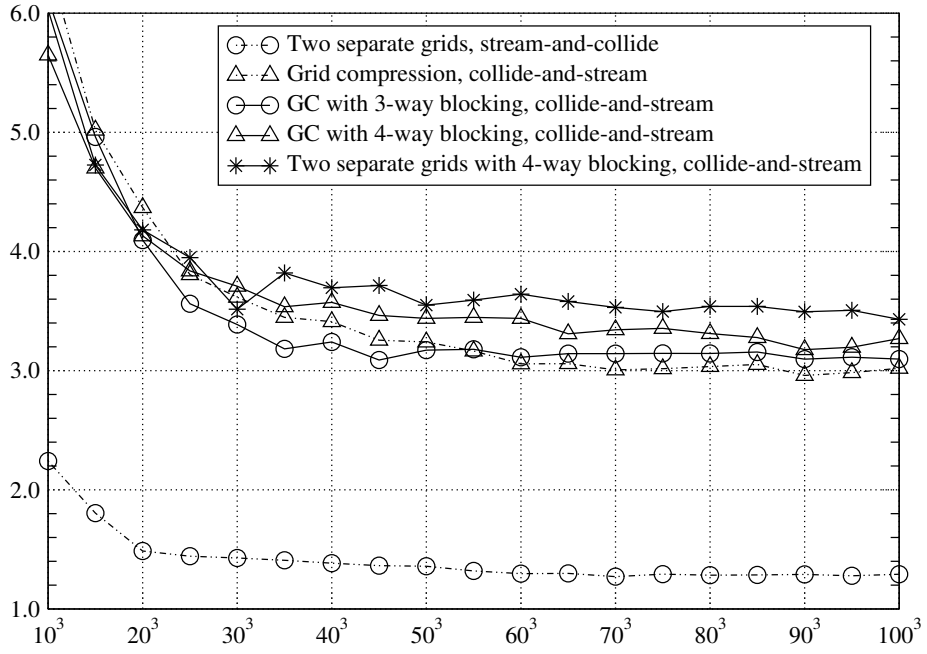


Figure 7.23: MLSUPS rates for various 3D LBM codes and problem sizes on Platform G.

Profiling data. Table 7.6 contains profiling data for Platform A[22]. The data refers to grids of size 100^3 and 16 time steps. Once more, we have used the profiling infrastructure DCPI [ABD+97]. The numbers in Table 7.6 show that, due to the enormous data set sizes occurring in 3D, our optimization techniques mainly improve the utilization of the relatively large L3 cache, while the L2 miss rate decreases only by about 30%. However, the data TLB miss rate increases by about a factor of 3 due to the fact that our blocking approaches require larger numbers of virtual pages to be accessed in an alternating fashion, see also Section 7.3.2.4.

A variety of further performance experiments concerning cache optimization techniques for the LBM in 3D can be found in [Igl03] and in more compact form in [PKW+03a, PKW+03b].

7.6 Summary of this Chaper

In this chapter, we have demonstrated that the application of suitably chosen data layout transformations and data access transformations (see the preceding Chapters 5 and 6) can improve the locality of data references and thus enhance the execution speed of grid-based numerical methods. We have examined multigrid and LBM implementations in both 2D and 3D.

A comparison of the performance results for the cases of 2D and 3D reveals that the MFLOPS rates (for the case of iterative linear solvers) and the MLSUPS rates (for the LBM case) are generally higher in 2D than in 3D. These differences are particularly dramatic for the multigrid case, see the performance results presented by Weiß in [Wei01]. In comparison with the theoretically available floating-point performance of today's CPUs, the MFLOPS rates that can be observed for the multigrid codes in 3D (even for the cache-optimized ones) must be considered disappointing, indicating that future work in this direction is required.

Moreover, the speedups that can be obtained for the 2D codes are generally more impressive, especially again in the multigrid case. Once more, we refer to the results presented in [Wei01]. Profiling experiments confirm that, in 3D, TLB misses may have a remarkable impact on the efficiency of the code since data TLBs are often too small to store a sufficiently large number of main memory addresses of virtual pages.

While, for example, highly optimized linear algebra kernels involving dense matrix computations — such as the codes included in the *LINPACK* benchmark suite [DBMS79] — typically run at considerable fractions of the peak speed [Don04], this is not the case for the implementations of grid-based iterative solvers. On the whole, their ratios of floating-point operations to data accesses (i.e., their computational intensities) are too low to avoid performance bottlenecks due to limited

[22] Compaq Tru64 UNIX V5.0A, Compaq cc V6.1, flags: `-O4 -fast -tune host -g3`

Hardware event (in 10^6)	Two grids, no blocking	GC, no blocking	Two grids, 4-way blocking	GC, 4-way blocking
L1 miss	272.7	276.8	278.3	294.4
L2 miss	407.0	413.4	308.1	272.5
L3 miss	121.7	45.8	43.9	25.8
TLB miss	1.5	1.1	8.6	4.9

Table 7.6: Hardware event counts for different 3D LBM codes on Platform A, measured with DCPI, problem size 100^3.

memory bandwidth and to keep the available functional units of the CPU constantly busy.

Furthermore, it can be observed that our LBM implementations usually exhibit a much higher floating-point performance (in terms of MFLOPS) than our multigrid codes. On Platform D for example, using the respective largest problem sizes, the most efficient cache-tuned 3D multigrid implementation runs at about 130 MFLOPS only (cf. Figure 7.7), while our fastest 3D LBM code runs at approximately 2.2 MLSUPS (cf. Figure 7.21), which corresponds to more than 550 MFLOPS. This behavior is explained by the increased computational intensity of the LBM. In 3D, for example, a Gauss-Seidel update step based on variable 7-point stencils involves 15 data accesses and 13 floating-point operations, resulting in a ratio of about 0.9 floating-point operations per data access, cf. (3.13) in Section 3.2.3. In contrast, in an implementation of the D3Q19 LBM model, the update of a single lattice site involves 38 data accesses and 255 floating-point operations, see Section 4.2.4. This yields a much larger ratio of about 6.7 floating-point operations per data access and thus illustrates the reduced pressure on the memory hierarchy in the LBM case.

Unfortunately, code optimization techniques may have conflicting goals. For example, loop blocking techniques, which target the optimization of the data cache utilization, often cause an increase in the number of alternatingly accessed pages such that the TLB miss rate raises considerably. Moreover, the application of loop blocking results in additional loops with relatively short iteration spaces, which increases the overall loop overhead. Depending on the properties of the branch prediction unit, blocking might further lead to an increase in the number of pipeline stalls due to incorrectly predicted branch targets. Finally, the more involved loop nests resulting from the application of loop blocking might prevent hardware prefetching units from working properly. As a consequence, optimizing code transformations often represent trade-offs and their impact on the overall efficiency of the code must always be examined carefully.

Part III

Designing Inherently Cache-Efficient Multigrid Algorithms

Chapter 8

Patch-Adaptive Multigrid

8.1 Overview

This chapter contains results of the ongoing research in our DiME project, cf. Section 1.2.4. In the following, we will present an approach towards novel *patch-adaptive multigrid algorithms* which are designed in order to exhibit a high potential of data locality and can thus be implemented cache-efficiently[1]. While the code optimizations from Chapters 5 and 6 target standard (multigrid) algorithms and aim at enhancing their cache utilization using techniques that may theoretically be automized, the approach we will present subsequently can be considered more fundamental. Our patch-adaptive multigrid algorithm is derived from the theory of the *fully adaptive multigrid method (FAMe)*, which has been introduced by Rüde [Rüd93a, Rüd93b].

The fully adaptive multigrid method is based on the integration of two different kinds of adaptivity into a single algorithmic framework. Firstly, *adaptive mesh refinement* is taken into account in order to reduce the number of degrees of freedom and, as a consequence, the computational work as well as the memory consumption of the implementation. For this purpose, Rüde has introduced the technique of *virtual global grids* [Rüd93b]. This approach can be seen as the combination of a recursive data structure representing an infinite sequence of uniformly refined meshes (each of which covers the entire computational domain) with the principle of lazy evaluation, which ensures that only a finite subset of grid nodes must actually be allocated and used during the computation in order to solve the problem up to the prescribed accuracy.

Secondly, the fully adaptive multigrid method employs the principle of *adaptive relaxation*, see Sections 8.2 and 8.3. With this update strategy, computational work can be focused on where it can efficiently improve the quality of the numerical approximation. Our patch-adaptive multigrid algorithm particularly exploits this second notion of adaptivity and therefore combines two essential advantages. First, it is inherently data-local and thus leads to a high utilization of the fast memory hierarchy levels, particularly the caches. This is primarily accomplished through the use of patch-based grid processing techniques, see Section 8.4 and particularly the explanation in Section 8.4.3.2.

Second, the robustness and, as a consequence, the numerical efficiency of our patch-adaptive multigrid algorithm is superior to that of classical multigrid methods, which we have presented in Section 3.3. This means that our novel algorithm can efficiently be applied to a wider class of problems. This benefit stems from the enhanced capabilities of the adaptive smoothers we employ.

[1] As we will describe in more detail in Section 8.4.2, a patch represents a suitably chosen (connected) block of the computational grid.

For details, we refer to Sections 8.2 and 8.3 below.

Similar multigrid techniques have already been studied in [Löt96, LR97]. However, the work presented in [Löt96, LR97] has focused on parallelization issues and the development of patch-adaptive mesh refinement, while our research primarily addresses the optimization of the cache performance through patch-based processing of the grid levels. Note that these design goals are by no means mutually exclusive. Instead, they can rather be considered independent of each other and future work should address their combined application.

The following description is related to the presentation given in [Chr03]. We primarily use the notation and the terminology introduced in [Rüd93b].

8.2 Adaptive Relaxation

8.2.1 Overview

As was mentioned above, the principle of adaptive relaxation is to concentrate computational work on those parts of the computational grid where it can efficiently reduce the error, thus reducing the total computational work needed to solve the problem. This is accomplished through the use of an *active set strategy*. In the course of the adaptive relaxation process, this active set is constantly being updated and used in order to guide the update order of the unknowns. In a multilevel context, one active set per grid level can be employed, see Section 8.3. We will restrict the following presentation to the case of *sequential (Gauss-Seidel-type)* adaptive relaxation. The principle of *simultaneous (Jacobi-type)* adaptive relaxation is explained in [Rüd93b] as well.

8.2.2 Definitions and Elementary Results

We consider the linear system (3.1) from Section 3.2.1.1. We assume that the matrix A is sparse and symmetric positive definite. As a common example, standard second-order finite difference discretizations of the (negative) Laplacian yield matrices with these properties.

Since A is symmetric positive definite, the problem defined by the linear system (3.1) can be rewritten as a minimization problem; solving (3.1) is equivalent to determining the unique vector $u^* = A^{-1}f \in \mathbf{R}^n$ which minimizes the discrete quadratic functional J, $J : \mathbf{R}^n \rightarrow \mathbf{R}$, which is defined as follows:

$$J(u) := \frac{1}{2}u^T Au + u^T f \ , \tag{8.1}$$

see [She94] for details and illustrations. Note that this equivalence is the motivation and the starting point for the algorithms belonging to the rich class of gradient methods such as the method of conjugate gradients, cf. Section 3.2.1.3. In general, it is helpful to keep this minimization problem in mind and to picture the iterative solution of (3.1) as the search for the minimum of J.

For the following presentation, we need to introduce the notion of the *scaled residual*. As usual, we assume that $u \in \mathbf{R}^n$ represents an approximation to the exact solution u^*. Recalling the definition of the residual r in (3.2) and using the notation from (3.11) (see again Section 3.2.1.3), the scaled residual $\bar{r} \in \mathbf{R}^n$ corresponding to the approximation u is defined as

$$\bar{r} := D^{-1}r = D^{-1}(f - Au) \ . \tag{8.2}$$

Finally, with $e_i \in \mathbf{R}^n$ standing for the i-th unit vector, the components θ_i, $1 \leq i \leq n$, of the scaled residual $\bar{r}$ are given by

$$\theta_i = \theta_i(u) := e_i^T \bar{r} \ . \tag{8.3}$$

With these definitions, an *elementary relaxation step* for equation i of the linear system (3.1), which updates the approximation $u \in \mathbf{R}^n$ and yields the new approximation $u' \in \mathbf{R}^n$, can be expressed as

$$u' = u + \theta_i e_i \ , \tag{8.4}$$

and the Gauss-Seidel update scheme given by (3.13) in Section 3.2.3 can be rewritten as follows:

$$\begin{aligned} u_i^{(k+1)} &= a_{i,i}^{-1} \left(f_i - \sum_{j<i} a_{i,j} u_j^{(k+1)} - \sum_{j>i} a_{i,j} u_j^{(k)} \right) \\ &= a_{i,i}^{-1} \left(f_i - \sum_{j<i} a_{i,j} u_j^{(k+1)} - \sum_{j\geq i} a_{i,j} u_j^{(k)} \right) + u_i^{(k)} \\ &= u_i^{(k)} + \theta_i \ . \end{aligned}$$

Here, θ_i represents the current scaled residual of equation i; i.e., the scaled residual of equation i *immediately* before this elementary update operation. As a consequence of this update step, the i-th component of the residual r and, obviously, the i-th component of the scaled residual $\bar{r}$ vanish.

The motivation of the development of the adaptive relaxation method is based on the following relation, which states how the algebraic error — cf. (3.3) in Section 3.2.1.3 — is reduced by performing a single elementary relaxation step:

$$||u^* - u||_A^2 - ||u^* - u'||_A^2 = a_{i,i}\theta_i(u)^2 \ . \tag{8.5}$$

As usual, $||v||_A$ stands for the (operator-dependent) *energy norm* of the vector $v \in \mathbf{R}^n$; i.e.,

$$||v||_A := \sqrt{v^T A v} \ .$$

Note that we have $v^T A v \geq 0$ for all $v \in \mathbf{R}^n$ since A is positive definite.

A proof of the previous lemma concerning the error reduction that results from an elementary update operation can be found in [Rüd93b]. Equation (8.5) implies that, with every elementary relaxation step, the approximation is either improved (if $\theta_i(u) \neq 0$) or remains unchanged (if $\theta_i(u) = 0$). Note that, since A is positive definite, all diagonal entries $a_{i,i}$ of A are positive. This follows immediately from $a_{i,i} = e_i^T A e_i$, $1 \leq i \leq n$. In other words, it is the positive definiteness of A which guarantees that the solution u will never become "worse" by applying elementary relaxation operations. See [She94] for examples and counter-examples.

Equation (8.5) represents the mathematical justification for the selection of the (numerically) most efficient equation update order. It states that the fastest error reduction results as long as those equations i are relaxed whose scaled residuals θ_i have relatively large absolute values. This is particularly exploited in the *method of Gauss-Southwell* where always the equation i with the largest value $|a_{i,i}\theta_i|$ is selected for the next elementary relaxation step. Yet, if no suitable update strategy is used, this algorithm will be rather inefficient since determining the equation whose residual has the largest absolute value is as expensive as relaxing each equation once [Rüd93b]. This observation motivates the principle of the adaptive relaxation method which is based on an *active set* of equation indices and can be interpreted as an algorithmically efficient approximation to the method of Gauss-Southwell.

In order that we can specify the handling of the active set and thus formulate the method of adaptive relaxation, it is convenient to introduce the *set of connections* of the grid node with index i, $1 \leq i \leq n$, as

$$\mathrm{Conn}(i) := \{j; 1 \leq j \leq n,\ j \neq i,\ a_{j,i} \neq 0\} \ . \tag{8.6}$$

Intuitively, the set $\mathrm{Conn}(i)$ contains the indices of those unknowns u_j that depend on u_i, except for u_i itself. Note that, due to the symmetry of A, these are exactly the unknowns u_j, $j \neq i$, that u_i depends on. The set $\mathrm{Conn}(i)$ thus contains the indices of the off-diagonal nonzeros of the i-th row of A.

Before we can provide an algorithmic formulation of the adaptive relaxation method, it is further necessary to introduce the *strictly active set* $S(\theta, u)$ of equation indices as follows:

$$S(\theta, u) := \{i;\ 1 \leq i \leq n,\ |\theta_i(u)| > \theta\}\ . \tag{8.7}$$

Hence, the set $S(\theta, u)$ contains the indices i of all equations whose current scaled residuals $\theta_i(u)$ have absolute values greater than the prescribed threshold $\theta > 0$. Alternative definitions of the strictly active set are discussed in [Rüd93b].

Finally, an *active set* $\tilde{S}(\theta, u)$ is defined as a superset of the strictly active set $S(\theta, u)$:

$$S(\theta, u) \subseteq \tilde{S}(\theta, u) \subseteq \{i;\ 1 \leq i \leq n\}\ . \tag{8.8}$$

Note that $\tilde{S}(\theta, u)$ may also contain indices of equations whose scaled residuals have relatively small absolute values. The idea behind the introduction of such an extended active set is that it can be maintained efficiently, see Algorithm 8.1. Implementations of the adaptive relaxation method are discussed in [Chr03, Dau01, WA99].

8.2.3 Algorithmic Description of the Adaptive Relaxation Method

Using the notation introduced previously, the core of the adaptive relaxation method is presented in Algorithm 8.1. Note that it is essential that the active set $\tilde{S}$ is initialized appropriately. Unless problem-specific information is known a priori, $\tilde{S}$ is initialized with all indices i, $1 \leq i \leq n$.

The algorithm proceeds as follows. Elementary relaxation steps are performed until the active set $\tilde{S}$ is empty, which implies that none of the absolute values of the scaled residuals θ_i exceeds the given tolerance θ anymore. In Steps 2 and 3, an equation index $i \in \tilde{S}$ is selected and removed from $\tilde{S}$. Only if the absolute value of the corresponding scaled residual θ_i is larger than θ (Step 4), equation i will be relaxed. The relaxation of equation i implies that all scaled residuals θ_j, $j \in \mathrm{Conn}(i)$, are changed, and it must therefore be checked subsequently if their absolute values $|\theta_j|$ have exceeded the tolerance θ. Hence, if equation i is relaxed (Step 5), the indices $j \in \mathrm{Conn}(i)$ will be put into the active set $\tilde{S}$ (Step 6), unless they are already contained in $\tilde{S}$.

Algorithm 8.1 Sequential adaptive relaxation.

Require: tolerance $\theta > 0$, initial guess $u := u^{(0)} \in \mathbf{R}^n$, initial active set $\tilde{S} := \tilde{S}^{(0)} \subseteq \{i;\ 1 \leq i \leq n\}$

1: **while** $\tilde{S} \neq \emptyset$ **do**
2: Pick $i \in \tilde{S}$
3: $\tilde{S} \leftarrow \tilde{S} \setminus \{i\}$
4: **if** $|\theta_i(u)| > \theta$ **then**
5: $u \leftarrow u + \theta_i(u) e_i$ *// Elementary relaxation step, cf.* (8.4)
6: $\tilde{S} \leftarrow \tilde{S} \cup \mathrm{Conn}(i)$
7: **end if**
8: **end while**

Ensure: $S = \emptyset$ *// The strictly (!) active set* S *is empty, see* (8.7)

Apparently, Algorithm 8.1 terminates as soon as the active set $\tilde{S}$ is empty. The minimization property resulting from the positive definiteness of A (cf. Section 8.2.2) guarantees that the algorithm will terminate for each initial guess $u^{(0)}$. After the algorithm has stopped and yielded the final approximation u, the absolute value of the scaled residual $\theta_i(u)$ of each equation i is less or equal than the tolerance θ; i.e., we have $|\theta_i(u)| \leq \theta$ for all i, $1 \leq i \leq n$. See [Rüd93b] for proofs and further details.

Considering Algorithm 8.1, the following observations can be made. Firstly, the selection of the equation to be relaxed next (Step 2) is done nondeterministically. According to [Rüd93b], this property might be exploited in order to implement locality-optimized update strategies. Note that, however, the data-local behavior of our patch-adaptive scheme results from the use of patches and not from exploiting this nondeterminism, cf. Section 8.4.

Secondly, in Step 6, indices of equations whose scaled residuals have relatively small absolute values may be inserted[2] into the active set $\tilde{S}$. Due to this property of the Algorithm 8.1 and due to $\tilde{S}$ being commonly initialized with all indices, it is important to check explicitly (Step 4) if the absolute value of the scaled residual $\theta_i(u)$ of equation i, which has been selected previously (Step 2), is actually greater than the tolerance θ. A more sophisticated activation approach, which determines the changes of the scaled residuals first before inserting the respective equation indices into $\tilde{S}$, will be discussed in Section 8.4 in the context of patch-oriented adaptive relaxation.

Thirdly, Step 4 of Algorithm 8.1 requires the computation of the scaled residual θ_i, which corresponds to the relaxation of equation i; recall (8.4) above. If $|\theta_i|$ is less or equal than the threshold θ, the result of this residual computation (i.e., the result of this elementary relaxation step) is discarded and the next equation is picked from the active set $\tilde{S}$ (Step 2). Note that this approach is necessary. If we actually did *not* discard this result, but executed the relaxation of equation i *without* putting its neighbors $j \in \mathrm{Conn}(i)$ into $\tilde{S}$, the absolute values of their scaled residuals θ_j could exceed the tolerance θ *without* the equations j being marked for further relaxation. Hence, the algorithm would not work properly since it would no longer be ensured that, after termination, we have achieved $|\theta_i| \leq \theta$, $1 \leq i \leq n$.

Finally, note that Algorithm 8.1 does not represent a linear iterative method, as defined in Section 3.2.1.3. This results from its adaptive update strategy which, for example, may cause some equations of the linear system (3.1) to be relaxed more often than others.

8.3 Adaptive Relaxation in Multigrid Methods

8.3.1 Overview

Generally speaking, when applied as a single-level solver, the adaptive relaxation method has the same asymptotic convergence behavior as classical iterative methods. While it reduces high-frequency error components efficiently, smooth error components are eliminated very slowly, cf. Section 3.3.4. In other words, due to the ill-conditioning of the system matrices under consideration, the norm of the (scaled) residual is reduced quickly, while the norm of the corresponding algebraic error may remain large and vanish only very slowly [Sch97]. Consequently, the adaptive relaxation method performs more efficiently than nonadaptive iterative schemes only as long as there are equations whose residuals have large absolute values; i.e., only in the beginning of the iteration [Rüd93b].

[2] Throughout this chapter, *inserting* an element x into a set S always refers to the operation $S \leftarrow S \cup \{x\}$. This implies that S remains unchanged if x is already contained in S.

The full advantage of the adaptive relaxation scheme becomes evident as soon as it is introduced into a multilevel structure. The idea is to employ the adaptive relaxation method as a smoother in a multigrid method, where it is not necessary to completely eliminate the error on the finer grids. Instead, it is enough to smooth the error such that a correction on a coarser grid can be computed efficiently. See again our overview of multigrid methods in Section 3.3.

From a mathematical point of view, the application of the adaptive relaxation method in the multigrid context corresponds to the approach of *local relaxation* [Rüd93b]. The principle of local relaxation states that the robustness of multigrid algorithms can be improved significantly by performing additional relaxation steps in the vicinity of singularities or perturbations from boundary conditions, for example. The purpose of these additional elementary relaxation steps is to enforce a sufficiently smooth error such that the subsequent coarse-grid correction performs efficiently. An analysis of this approach is beyond the scope of this thesis. Instead, we refer to [Rüd93b] and especially to the references provided therein.

In the following, we will briefly discuss the theory which is needed in order to interpret multigrid methods as iterative schemes on an extended version of the original linear system (3.1) from Section 3.2.1.1. This theory, which has been introduced by Griebel in [Gri94a, Gri94b], is needed to motivate and to explain our approach towards inherently cache-conscious multigrid algorithms, see also [Rüd93b].

8.3.2 Preparations

We use indices $l = 0, 1, \ldots, L$ in order to denote the individual levels of the multigrid hierarchy. The numbers $n_0, n_1, \ldots, n_L$ represent the dimensions of the vector spaces, $n_j \leq n_{j+1}$, $0 \leq j < L$, where $n = n_L$ stands for the dimension of the original linear system (3.1). Furthermore, we declare

$$\hat{n} := \sum_{l=0}^{L} n_l \ .$$

This means that $\hat{n}$ denotes the number of degrees of freedom contained in the entire hierarchy of grid levels.

The linear systems corresponding to the individual grid levels are given by

$$A_l u_l = f_l \ ,$$

where $0 \leq l \leq L$. Hence, the hierarchy comprises $L + 1$ grid levels. The finest level representing the original system to be solved is characterized by the level index $l = L$, and the coarsest level is characterized by the level index $l = 0$. We define $A_L := A$, and $f_L := f$, cf. again (3.1).

The restriction operators and the interpolation operators are given by

$$I_{l+1}^{l} : \mathbf{R}^{n_{l+1}} \rightarrow \mathbf{R}^{n_l} \ ,$$

as well as

$$I_{l}^{l+1} : \mathbf{R}^{n_l} \rightarrow \mathbf{R}^{n_{l+1}} \ ,$$

respectively, where $0 \leq l < L$. We further assume that, for $0 \leq l < L$, the relation

$$I_{l}^{l+1} = \left(I_{l+1}^{l}\right)^T \tag{8.9}$$

holds and that the coarse grid matrices A_l are computed recursively using the Galerkin products:

$$A_l := I_{l+1}^{l} A_{l+1} I_{l}^{l+1} \ ,$$

cf. Section 3.3.5.3. Note that, in this setting, the introduction of a scaling factor $c_l^{l+1} \neq 0$ in (8.9) — as is the case in (3.23) — would lead to a corresponding scaling of the coarse-grid operator A_l, but would not change the results of the coarse-grid correction.

Again, we assume the matrix A_L corresponding to the finest level of the hierarchy to be symmetric positive definite. According to the discussion concerning the choice of coarse-grid operators in Section 3.3.5.3, the resulting coarse-grid matrices A_l, $0 \leq l < L$, are symmetric positive definite as well. Hence, each of the system matrices is regular and each linear system of the hierarchy is equivalent to a minimization problem, cf. Section 8.2.2.

In order to simplify the presentation, it is convenient to define for any pair of level indices l_1 and l_2 ($0 \leq l_1 < l_2 \leq L$) the product $I_{l_2}^{l_1}$, $I_{l_2}^{l_1} : \mathbf{R}^{n_{l_2}} \to \mathbf{R}^{n_{l_1}}$, of restriction operators as follows:

$$I_{l_2}^{l_1} := I_{l_1+1}^{l_1} \cdots I_{l_2-1}^{l_2-2} I_{l_2}^{l_2-1} \ .$$

Correspondingly, the product $I_{l_1}^{l_2}$, $I_{l_1}^{l_2} : \mathbf{R}^{n_{l_1}} \to \mathbf{R}^{n_{l_2}}$, of interpolation operators is given by

$$I_{l_1}^{l_2} := \left(I_{l_2}^{l_1}\right)^T = I_{l_2-1}^{l_2} I_{l_2-2}^{l_2-1} \cdots I_{l_1}^{l_1+1} \ . \tag{8.10}$$

In order to simplify the presentation, I_l^l stands for the identity in $\mathbf{R}^{n_l}$.

8.3.3 The Extended Linear System

8.3.3.1 Nonunique Representation of Fine-Grid Vectors

The key observation is that any vector $u \in \mathbf{R}^n$ (i.e., any grid function on the finest level of the multigrid hierarchy) can be represented *nonuniquely* as

$$u = \sum_{l=0}^{L} I_l^L u_l \ , \tag{8.11}$$

where $u_l \in \mathbf{R}^{n_l}$, $0 \leq l \leq L$. Recall that the interpolation operator I_l^L maps a grid function $u_l \in \mathbf{R}^{n_l}$ from any level l to the finest level L, $0 \leq l \leq L$. In [Gri94b], this situation is represented geometrically by considering the nodal basis functions corresponding to each level of the grid hierarchy. The grid functions u_l stored on the coarser grid levels l, $0 \leq l < L$, can be interpreted either as actual *contributions* or as *corrections* to the grid function $u \in \mathbf{R}^n$ to be represented. Hence, in a certain sense, these considerations present a unified interpretation of the multigrid CGC scheme and the multigrid FAS method.

We introduce the operator $\hat{I}$, $\hat{I} : \mathbf{R}^n \to \mathbf{R}^{\hat{n}}$, as

$$\hat{I} := \begin{bmatrix} I_L^0 \\ I_L^1 \\ \vdots \\ I_L^L \end{bmatrix} \in \mathbf{R}^{\hat{n} \times n} \ .$$

Since I_L^L stands for the identity matrix in $\mathbf{R}^{n \times n}$, we have $\hat{I}v = 0$ only if $v = 0$. Hence, we obtain

$$\dim \ker \hat{I} = 0 \ . \tag{8.12}$$

We will refer to this property below. Furthermore, using (8.10), we find that the transpose $\hat{I}^T$, which defines a mapping $\hat{I}^T : \mathbf{R}^{\hat{n}} \to \mathbf{R}^n$, can be written as

$$\hat{I}^T = \begin{bmatrix} I_0^L & I_1^L & \ldots & I_L^L \end{bmatrix} \in \mathbf{R}^{n \times \hat{n}} \ .$$

If we declare the vector $\hat{u} \in \mathbf{R}^{\hat{n}}$ as

$$\hat{u} := \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_L \end{bmatrix} \quad , \tag{8.13}$$

$u_l \in \mathbf{R}^{n_l}$, the multi-scale representation of u given by (8.11) can be rewritten in a simpler form:

$$u = \hat{I}^T \hat{u} \quad . \tag{8.14}$$

Obviously, we have

$$\dim \operatorname{im} \hat{I}^T = n \quad . \tag{8.15}$$

From this, it follows that

$$\dim \ker \hat{I}^T = \hat{n} - n \quad . \tag{8.16}$$

This means that, from an algebraic viewpoint, the additional degrees of freedom resulting from the introduction of the coarser grid levels contribute to the null space of the operator $\hat{I}^T$. Using the geometric motivation from [Gri94b], the additional nodal basis functions corresponding to the coarser grid levels l, $0 \leq l < L$, do not increase the space of functions that can be represented on the finest grid level L. However, the introduction of the nodal basis functions on the coarser grids allows grid functions on the finest grid level to be represented nonuniquely, cf. (8.11). We will also refer to this observation below.

8.3.3.2 Derivation of the Extended Linear System

With the previous notation at hand, the minimization of the functional J from (8.1) can formally be rewritten as

$$\min_{\substack{u_l \in \mathbf{R}^{n_l} \\ 0 \leq l \leq L}} \left[\frac{1}{2} \left(\sum_{l=0}^{L} I_l^L u_l \right)^T A \left(\sum_{l=0}^{L} I_l^L u_l \right) - \left(\sum_{l=0}^{L} I_l^L u_l \right)^T f \right] \quad .$$

From (8.14), it follows that this is equivalent to the minimization of the corresponding extended discrete quadratic functional $\hat{J}$, $\hat{J} : \mathbf{R}^{\hat{n}} \to \mathbf{R}$, which is given by

$$\hat{J}(\hat{u}) := \frac{1}{2} \hat{u}^T \left(\hat{I} A \hat{I}^T \right) \hat{u} - \hat{u}^T \left(\hat{I} f \right) \quad . \tag{8.17}$$

So far, this situation can be pictured as follows. There is a unique vector $u \in \mathbf{R}^n$ which minimizes the functional J defined in (8.1). This vector corresponds to the unique solution u^* of the linear system (3.1) from Section 3.2.1.1, which represents the finest level of the multigrid hierarchy. However, there are infinitely many vectors $\hat{u} \in \mathbf{R}^{\hat{n}}$ such that $u^* = \hat{I}^T \hat{u}$ holds. In other words, the problem of minimizing the extended functional $\hat{J}$ has infinitely many solutions.

In order that a vector $\hat{u} \in \mathbf{R}^{\hat{n}}$ minimizes $\hat{J}$, it is necessary that the first partial derivatives of $\hat{J}$ vanish in $\hat{u}$; i.e., that $\nabla \hat{J}(\hat{u}) = 0$ holds. From this requirement and with the definitions

$$\hat{A} := \hat{I} A \hat{I}^T \in \mathbf{R}^{\hat{n} \times \hat{n}}$$

as well as

$$\hat{f} := \hat{I} f \in \mathbf{R}^{\hat{n}} \quad ,$$

we obtain the *extended linear system*

$$\hat{A} \hat{u} = \hat{f} \quad , \tag{8.18}$$

see [Rüd93b]. The extended system matrix $\hat{A}$ can be represented as

$$\hat{A} = \begin{bmatrix} A_0 & I_1^0 A_1 & \dots & \dots & I_L^0 A_L \\ A_1 I_0^1 & A_1 & I_2^1 A_2 & \dots & I_L^1 A_L \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_L I_0^L & \dots & \dots & A_L I_{L-1}^L & A_L \end{bmatrix} , \tag{8.19}$$

and the right-hand side $\hat{f}$ of the extended linear system is given by

$$\hat{f} = \begin{bmatrix} I_L^0 f \\ I_L^1 f \\ \vdots \\ f \end{bmatrix} . \tag{8.20}$$

The linear system (8.18) covers the entire multigrid hierarchy. For each node i belonging to grid level l_i, the corresponding row of the matrix $\hat{A}$ contains the entries which specify how the solution value at node i depends on the values at its neighboring nodes belonging to the *same* grid level (given by the respective entries of the diagonal block A_{l_i}) and on values at nodes belonging to *different* grid levels (given by the off-diagonal blocks of $\hat{A}$). In particular, the entries of the subdiagonal blocks of $\hat{A}$ represent the influence of nodes located on the respective coarser levels $l < l_i$, while the entries of the superdiagonal blocks of $\hat{A}$ represent the influence of nodes belonging to the respective finer levels $l > l_i$.

8.3.3.3 Properties of the Matrix of the Extended Linear System

The matrix $\hat{A}$ has several properties to be observed at this point. Firstly, $\hat{A}$ is symmetric since A is symmetric and we therefore find that

$$\hat{A}^T = \left(\hat{I} A \hat{I}^T\right)^T = \hat{I} A \hat{I}^T = \hat{A} \ .$$

Secondly, $\hat{A}$ is singular with

$$\dim \ker \hat{A} = \hat{n} - n \ .$$

This follows from (8.16), from the regularity of the matrix A of the original system, and from (8.12). More precisely, we have

$$\ker \hat{A} = \ker \hat{I}^T \ ,$$

see [Gri94b]. Thirdly, $\hat{A}$ is positive semidefinite; i.e., we have $\hat{u}^T \hat{A} \hat{u} \geq 0$ for all vectors $\hat{u} \in \mathbf{R}^{\hat{n}}$. This property follows from

$$\hat{u}^T \hat{A} \hat{u} = \left(\hat{I}^T \hat{u}\right)^T A \left(\hat{I}^T \hat{u}\right)$$

with both the singularity of $\hat{I}^T$ and the positive definiteness of the matrix A itself. The positive semidefiniteness of $\hat{A}$ reflects that each solution of the extended linear system (8.18) minimizes the extended quadratic functional $\hat{J}$ from (8.17), and vice versa.

Note that the extended linear system (8.18) is actually solvable; i.e., we have

$$\hat{f} \in \mathrm{im} \hat{A} \ . \tag{8.21}$$

This can be seen as follows. We can represent the unique solution u^* of (3.1) nonuniquely as $u^* = \hat{I}^T \hat{u}'$, since we know from (8.15) that $\hat{I}$ defines a surjective mapping. Hence, we obtain

$$A u^* = A \left(\hat{I}^T \hat{u}'\right) = f \ ,$$

which in turn implies that

$$\hat{A}\hat{u}' = \left(\hat{I}A\hat{I}^T\right)\hat{u}' = \hat{I}A\left(\hat{I}^T\hat{u}'\right) = \hat{I}f = \hat{f} \ .$$

From this, (8.21) follows.

An analysis of the spectral properties of the extended matrix $\hat{A}$ reveals that the (generalized) condition number of $\hat{A}$ is of order $\mathcal{O}(1)$, as soon as an appropriate preconditioner is additionally introduced. A detailed presentation of this theory exceeds the scope of this thesis and can be found in [Rüd93b] and in [Gri94b]. On the whole, this result illustrates that multigrid methods can reduce the algebraic error uniformly over the whole frequency spectrum, avoiding the problems which commonly arise when basic iterative methods — such as the ones presented in Section 3.2 — are applied to ill-conditioned linear systems, cf. Section 3.3.4. In fact, this uniform error reduction is accomplished through the integration of the coarser scales.

For a discussion of further properties of the extended linear system (8.18) and more details concerning the interpretation of the abstract theory, see [Gri94b].

8.3.4 Iterative Solution of the Extended Linear System

The application of iterative algorithms to the extended linear system (8.18) represents the starting point for alternative approaches towards the analysis and the development of multigrid methods. Griebel has shown how conventional multigrid algorithms can be interpreted as iterative methods applied to the extended linear system (8.18), which implicitly covers the whole grid hierarchy. Conversely, it has further been examined how standard iterative schemes applied to (8.18) can be interpreted as multigrid methods [Gri94b]. A more compact presentation of these results is provided in [Gri94a]. See [BP94] for general convergence results on iterative methods applied to linear systems with singular matrices.

As was mentioned in Section 8.1, in addition to the concept of virtual global grids, the fundamental principle of the fully adaptive multigrid method is the application of the adaptive relaxation scheme to the extended system (8.18) in order to efficiently reduce the scaled residuals on all levels of the grid hierarchy. In summary, the advantage of the fully adaptive multigrid method can thus be stated as follows: it maintains the property of efficient residual reduction (cf. Section 8.2), but it avoids the problems of stalling convergence related to the ill-conditioning of the original linear system (3.1), cf. Section 8.3.3.3.

Note that, since the active set contains equations corresponding to all levels of the grid hierarchy, nonstandard cycling strategies can be derived. Such strategies are not necessarily based on a level-oriented processing of the grid hierarchy, as is the case for the classical multigrid schemes presented in Section 3.3. See [Rüd93b] for details.

8.4 Introducing Patch Adaptivity

8.4.1 Overview of the Patch-Adaptive Multigrid Scheme

The subsequent discussion will be closely adapted to the presentation provided in [Gri94b, §5]. Our patch-adaptive multigrid method represents a level-oriented iterative scheme for the extended linear system (8.18). In other words, it is based on a block partitioning of the extended linear system (8.18), where the blocks are defined by the individual grid levels of the hierarchy.

More precisely, our method is characterized by the combination of an outer block Gauss-Seidel iteration (visiting the individual grid levels of the hierarchy) with an inner (inexact) iterative scheme applied to each individual block. On the coarsest block (i.e., on the block corresponding to the coarsest level of the grid hierarchy), a direct method based on a triangular factorization of the corresponding system matrix A_0 can be employed instead. In this case, the corresponding linear systems are solved exactly, see Section 3.2.1.2. As the inner iterative scheme on the relatively coarse levels, standard smoothers are used (e.g., the method of Gauss-Seidel). On all remaining grid levels which are sufficiently fine, we employ the patch-based adaptive smoother which will be described below.

The algorithms which have been implemented cover V-cycle iterations as well as the FMG method based on V-cycles, cf. Section 3.3.5.8. However, for ease of presentation, we will focus on the description of the V-cycle scheme only. The outer iteration is then given by a symmetric block Gauss-Seidel method; it starts with the finest grid level ($l = L$), moves down to the coarsest grid level ($l = 0$), and then traverses the hierarchy back up to the finest grid level again.

For each grid level l, $0 \leq l \leq L$, the corresponding linear system is given by

$$A_l u_l = f_l - \sum_{j=0}^{l-1} A_l I_j^l u_j - \sum_{j=l+1}^{L} I_j^l A_j u_j \ . \tag{8.22}$$

These equations follow from the representations of the extended system matrix $\hat{A}$, the extended vector $\hat{u}$ of unknowns, and the extended right-hand side $\hat{f}$ in (8.19), (8.13), and (8.20), respectively.

Griebel has demonstrated that, in the course of standard multigrid CGC V-cycles, the linear systems given by (8.22) are relaxed whenever level l, $l > 0$, is visited. For $l = 0$ (i.e., whenever the coarsest level of the hierarchy is visited), (8.22) is typically solved exactly [Gri94b].

The first sum $\sum_{j=0}^{l-1} A_l I_j^l u_j$ occurring on the right-hand side of (8.22) for level l, $l > 0$, is implicitly accounted for by immediately adding the corrections from the coarser levels to the solutions stored on the respective finer grids, cf. Step 10 of Algorithm 3.3 in Section 3.3.5.5. Correspondingly, the first term f_l as well as the second sum $\sum_{j=l+1}^{L} I_j^l A_j u_j$ are implicitly accounted for by defining the current right-hand side of each level l, $l < L$, as the restricted residual of the corresponding next finer level $l + 1$, cf. Step 3 of Algorithm 3.3.

In the case of our patch-adaptive multigrid V-cycle method, this outer iteration remains unchanged. However, we have developed a patch-adaptive and inherently cache-aware inner iterative scheme in order to smooth the algebraic errors on the respective fine grid levels; i.e., on those grid levels which are fine enough such that patch processing is reasonable. As a consequence of the adaptive processing approach, our multigrid V-cycle algorithm parallels the fully adaptive multigrid method. Since our scheme employs a conservative cycling strategy involving a level-oriented processing of the grid hierarchy, it requires that we maintain one active set for each grid level on which the patch-adaptive smoother is employed.

Note that an alternative patch-oriented multigrid smoother has been proposed and examined in [Löt96] and in more compact form in [LR97]. The respective numerical results have recently been confirmed in [Chr03]. Instead of implementing an active set strategy, this smoother is based on a fixed update sequence of the patches. Whenever a patch is visited, an empirically chosen, fixed number of relaxation sweeps is performed on it. In contrast to our algorithm, this scheme does not represent an adaptive method which guarantees an upper bound for the resulting scaled residual $\bar{r}$. However, if the patch sizes are chosen appropriately, it can be considered inherently cache-conscious as well.

For the sake of simplicity, we will subsequently focus on the case of 2D rectangular structured grids only. However, our patch-adaptive approach is by no means restricted to structured grids and problem domains with simple geometries. Furthermore, the extension to 3D is straightforward.

We assume that the discrete operator $A = A_L$ corresponding to finest level L of the grid hierarchy is characterized by compact 9-point stencils. With the assumptions from Section 8.3.2, this implies that the coarse-grid operators, each of which is computed as the Galerkin product of the corresponding fine-grid matrix and the inter-grid transfer operators, are given by compact 9-point stencils as well. The local structure of the linear equations on all grid levels (i.e., the compactness of the discretization stencils) is crucial for the patch-adaptive relaxation scheme[3].

8.4.2 Patch Design

The adaptive relaxation scheme which we propose is based on *patches*. A patch can be interpreted as a frame which specifies a region of a computational grid. We only consider the case of non-overlapping patches. In particular, our approach is based on partitionings of the individual 2D grids of the hierarchy into rectangular tiles.

From a theoretical viewpoint, in order to optimize the numerical efficiency of the patch-based adaptive relaxation method, it is by no means required that all patches belonging to a grid level have the same size. Instead, small patches can be chosen in the vicinity of singularities and perturbations from boundary conditions while larger patches can be located where the solution is supposed to be smooth. However, the integration of such adaptively refined patch structures complicates the algorithm (particularly, the activation of the neighboring patches) and will be omitted for the sake of simplicity. Note that patch sizes may vary from grid level to grid level.

In the following, the set of patches corresponding to level l will be denoted as $\mathcal{P}_l$. Furthermore, we will use the notation $i \in P$ in order to express that grid node i is located on patch P.

Figure 8.1 shows the structure of an individual patch P, which represents an *interior patch* and therefore has eight neighbors. Patches located close to the boundary of the grid have less than eight neighbors and are called *boundary patches*. The possible directions where neighboring patches may be located are specified in Figure 8.2. Correspondingly, line patches in 1D have at most two neighbors, whereas cubic patches in 3D have up to 26 neighbors. The number of unknowns corresponding to patch P is denoted as n_P. Since we use rectangular 2D patches only, each patch is characterized by two pairs $(x_{\min}, x_{\max})$ and $(y_{\min}, y_{\max})$ of indices. As is illustrated in Figure 8.1, these indices specify the grid nodes located in the patch corners.

For ease of presentation, Figure 8.1 shows a relatively small interior patch P of 8×8 grid nodes only; i.e., $n_P = 64$. From the viewpoint of computer architecture, the sizes of the patches should be chosen according to the capacity of the cache for which the code is tailored. For example, a patch of 32×32 grid nodes occupies $32 \cdot 32 \cdot (9 + 1 + 1)$ floating-point numbers. Recall that, at each interior grid node i, we need to store a stencil of nine coefficients (i.e., the entries $(A_l)_{i,j}$ of the i-th row of the corresponding system matrix A_l), the right-hand side $(f_l)_i$, as well as the current approximation $(u_l)_i$. If double precision values are used, this amounts to 88 kB of memory. Note that the introduction of suitable array paddings may be essential in order that cache conflicts are eliminated and that the patch-based codes actually perform cache-efficiently, cf. Section 5.2.

[3]Recall that the local structure of the stencils is also necessary in order that most of the data locality optimizations from Chapters 5 and 6 can be applied.

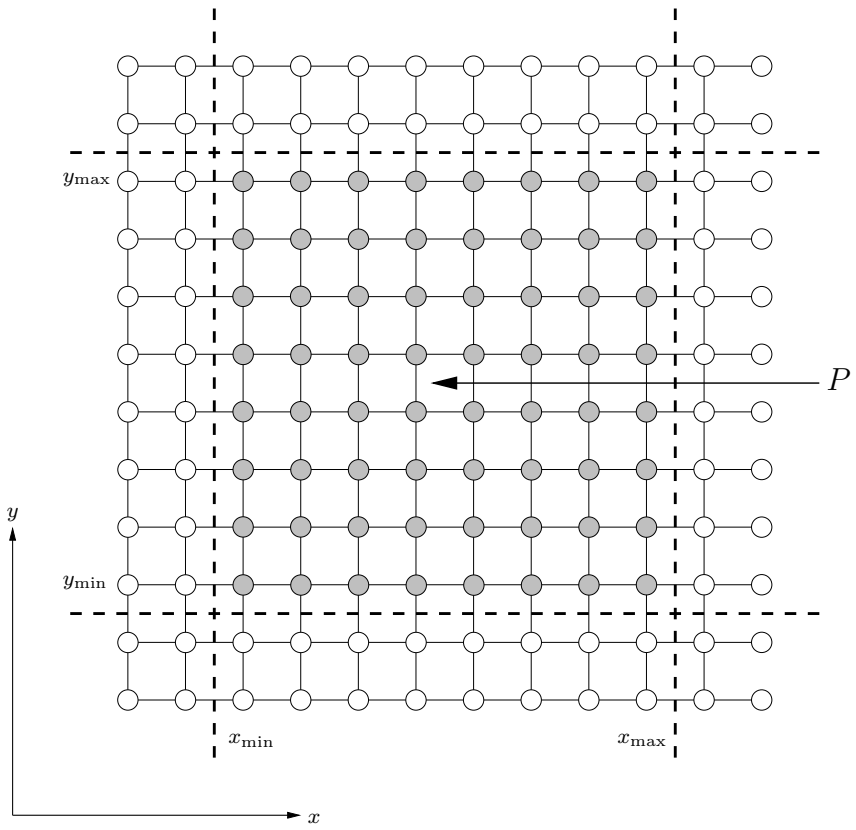


Figure 8.1: Patch structure and patch dimensions.

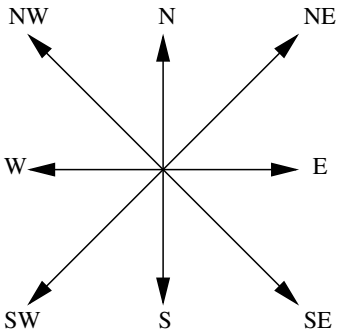


Figure 8.2: Directions of neighboring patches.

8.4.3 Immediate Consequences of the Patch-Adaptive Approach

8.4.3.1 Reduction of Granularity and Overhead

In the case of the patch-adaptive iteration on grid level l, the corresponding active set $\tilde{S}_l$ does not contain the indices of individual equations (i.e., the indices of individual grid nodes), as is the case for the point-based adaptive relaxation scheme presented in Section 8.2. Instead, $\tilde{S}_l$ contains the indices of patches. Note that we will omit the level index, if it is not necessary.

Yet, on each respective grid level l, the active set $\tilde{S}_l$ is a superset of the strictly active set S_l, as required by Definition (8.8) in Section 8.2.2. Accordingly, the active set $\tilde{S}_l$ must be initialized appropriately. Unless problem-specific information is known, $\tilde{S}_l$ is initialized with all patches corresponding to level l; i.e, we typically set $\tilde{S}_l := \mathcal{P}_l$ in the beginning.

More sophisticated variants of the patch-based multigrid scheme could be derived from the theory of the fully adaptive multigrid method, dispense with the level-wise processing of the grid hierarchy, and employ adaptive cycling strategies instead. In such cases, a single active set containing patches on all corresponding grid levels might be sufficient. The investigation of such algorithms exceeds the scope of this thesis and is left to future work, see Section 10.3.3.

As a consequence of our approach, each patch represents an indivisible entity which can only be inserted into the active set and processed as a whole. Since the number of patches is usually much smaller than the number of unknowns (i.e., we commonly have $|\mathcal{P}_l| << n_l$), the granularity of the adaptive relaxation scheme and, in addition, the overhead of maintaining the active set due to the selection, removal, and insertion of patches are both reduced significantly.

8.4.3.2 Increased Update Rate

Our patch-based approach represents a trade-off between computational work and execution speed. On one hand, in comparison with the original point-based adaptive relaxation scheme from Section 8.2, a higher number of elementary relaxation steps may be required to reduce the (scaled) residual of an approximation on a grid level below some given tolerance. For the definition of an elementary relaxation step, we refer to (8.4) in Section 8.2.2.

On the other hand, our patch-oriented approach causes a significant increase in the rate of elementary relaxation steps; i.e., in the MFLOPS rate. There are two reasons for the enhanced update rate. Firstly, the reduced overhead due to the simplified maintenance of the active set immediately implies a larger number of elementary relaxation steps per time unit. Note that, with the patch-based approach, it is not required to update the active set after each elementary relaxation step.

Secondly, according to Section 8.3.2, each system matrix A_l, $0 \leq l \leq L$, is symmetric positive definite. Hence, the theory from Section 8.2.2 can be applied, and we are free to choose the order of the elementary relaxation steps. Therefore, we may relax a patch repeatedly once it has been loaded into the cache. The resulting improved utilization of cache contents causes a further speedup of the update rate. This second aspect has already been examined in [Löt96, LR97]. In particular, it has been demonstrated that, once an appropriately sized patch has been loaded into the cache and updated for the first time, the costs of subsequently relaxing it a few more times are relatively low. According to the experiments presented in [Löt96], each further relaxation sweep has taken about 25% of the execution time of the first one. These results again illustrate our general initial observation that moving data has become much more expensive than processing data, see again Section 2.2.2.

8.4.4 Patch Processing

In order that we can describe our patch-adaptive relaxation scheme, it is necessary to explain initially what it means to determine the *scaled patch residual* and to *relax a patch*.

8.4.4.1 Definition of the Scaled Patch Residual

As is the case for the point-based adaptive relaxation scheme presented in Algorithm 8.1, the objective of the patch-based method is to iterate on the approximation u_l stored on the corresponding grid level l until none of the absolute values of the scaled residuals θ_i, $1 \leq i \leq n_l$, exceeds the prescribed tolerance $\theta > 0$. Note that, hereby, θ may depend on the level index l; i.e., $\theta = \theta_l$.

In order to simplify the subsequent presentation, we use $\bar{r}_P \in \mathbf{R}^{n_P}$ to denote the *scaled patch residual* of patch $P \in \mathcal{P}_l$. The vector $\bar{r}_P$ stands for the restriction of the overall scaled residual $\bar{r}_l \in \mathbf{R}^{n_l}$, which corresponds to the current approximation u_l on level l, to those unknowns i with $i \in P$. See again the definition of the scaled residual given by (8.2) in Section 8.2.2.

The calculation of $\bar{r}_P$ is demonstrated in Algorithm 8.2. The vector $\bar{r}_P$ is successively filled with the appropriate components of the (global) scaled residual $\bar{r}$. This requires the use of the additional counter variable j.

Since we require the absolute value of the scaled residual θ_i at each node i to fall short of the tolerance θ, we use the *maximum norm* $||\cdot||_\infty$ in the following. Recall that, for the vector $\bar{r}_P \in \mathbf{R}^{n_P}$, its maximum norm is defined as

$$||\bar{r}_P||_\infty := \max_{1 \leq j \leq n_P} \left| (\bar{r}_P)_j \right| = \max_{1 \leq j \leq n_P} |\theta_j| \ ,$$

cf. [Sch97], for example.

8.4.4.2 Patch Relaxation

In the course of the patch-adaptive iteration, patches will repeatedly be *relaxed*. When patch P is relaxed, elementary relaxation steps are applied to the equations i belonging to P. Hereby, a fixed ordering of these n_P equations is assumed, see Algorithm 8.3.

On one hand, cache efficient execution can only be expected, if a patch (with an appropriately chosen size) is relaxed at least twice. On the other hand, it is generally not reasonable to apply too many patch relaxations since, in a multigrid setting, we only need to smooth the algebraic errors on the finer grids.

Applying a large number of patch relaxations corresponds to the computation of a relatively accurate solution on the current patch. Hereby, the neighboring patches impose Dirichlet-type boundary conditions on the current patch which may be modified further on. This parallels the application of a single-level domain decomposition approach [QV99].

Algorithm 8.2 Computing the scaled residual $\bar{r}_P$ of patch $P \in \mathcal{P}_l$.

Require: current approximation $u_l \in \mathbf{R}^{n_l}$

1: $j = 1$
2: **for** each equation index $i \in P$ **do**
3: $\quad (\bar{r}_P)_j = (A_l)_{i,i}^{-1} \, e_i^T \, (f_l - A_l u_l)$ // *Right-hand side equals* $\theta_i(u_l)$
4: $\quad j = j + 1$
5: **end for**

Algorithm 8.3 Relaxation of patch $P \in \mathcal{P}_l$.

Require: current approximation $u_l \in \mathbf{R}^{n_l}$

1: **for** each equation index $i \in P$ **do**
2: *// Apply elementary relaxation step, cf.* (8.4)*:*
3: relax(i)
4: **end for**

8.4.5 Activation of Neighboring Patches

8.4.5.1 Problem Description

We assume that the patch-adaptive relaxation method is currently processing grid level l. For ease of notation, we omit the level index l. Hence, the approximation on the current level l is denoted as $u \in \mathbf{R}^{n_l}$, and the components of the approximation u are represented by the values u_i, $1 \leq i \leq n_l$.

In order that the patch-adaptive scheme works correctly, we must ensure that each patch $P \in \mathcal{P}_l$, which covers at least one node i with $|\theta_i| > \theta$, is an element of the active set $\tilde{S} = \tilde{S}_l$ corresponding to level l. This requirement guarantees that, once $\tilde{S}$ is empty and the algorithm therefore has terminated and yielded the approximate solution u, we have $|\theta_i(u)| \leq \theta$ for all i, $1 \leq i \leq n_l$.

Assume that we have just relaxed patch $P \in \mathcal{P}_l$. This means that we have updated those components u_i of the approximation u with $i \in P$. Consequently, we have to decide which of the neighboring patches of P must be inserted into the active set $\tilde{S}$ and thus marked for further processing. Assume that patch $P' \in \mathcal{P}_l$ is a neighboring patch of P. In general, the previous relaxation of P has influenced the scaled residuals θ_j at all nodes $j \in P'$, which have neighbors $i \in P \cap \mathrm{Conn}(j)$.

In the following, we will first present a simple but useful theoretical observation. Afterwards, we will discuss alternative activation policies for our patch-adaptive relaxation scheme.

8.4.5.2 Changes in Scaled Residuals

In general, if the approximation u_i stored at node i is changed by the value δ_i (i.e., $u_i \to u_i + \delta_i$), and if $j \in \mathrm{Conn}(i)$, the scaled residual θ_j at node j changes by the value

$$\delta\theta_j := \frac{a_{j,i}\delta_i}{a_{j,j}} \ ;$$

i.e., $\theta_j \to \theta_j + \delta\theta_j$. This follows immediately from the definition of the scaled residual θ_j, see (8.2) and (8.3) in Section 8.2.2. Recall that each system matrix A_l of the grid hierarchy is symmetric positive definite, which implies that $a_{j,j} > 0$, $1 \leq j \leq n_l$.

The preceding observation can be used in order to define sophisticated *relaxation/activation policies*.

8.4.5.3 Relaxation/Activation Policies

In order that the absolute value of each scaled residual θ_i, $1 \leq i \leq n_l$, finally falls below the prescribed tolerance $\theta > 0$, a variety of (more or less sophisticated) relaxation/activation policies can be derived from the point-based version of the adaptive relaxation scheme. In the following, we will present three alternative approaches. Each of them is motivated by the observation that a few additional relaxation sweeps over the current patch are both reasonable (in terms of smoothing the algebraic error) and relatively inexpensive (in terms of code execution time) as soon as the patch data has been loaded into cache, cf. Section 8.4.3.2.

Approach 1. This approach is very closely adapted to the point-based adaptive relaxation scheme from Section 8.2. As long as the active set $\tilde{S}$ is not empty yet, a patch P is picked and removed from $\tilde{S}$. We distinguish two cases:

1. If $||\bar{r}_P||_\infty \leq \theta$, the next patch is selected and removed from $\tilde{S}$, and the algorithm continues.

2. If $||\bar{r}_P||_\infty > \theta$, P is relaxed an empirically chosen, fixed number of times and, afterwards, the neighbors of P as well as the patch P itself are inserted into the active set $\tilde{S}$. Note that it is essential to insert P into $\tilde{S}$ again, since we do not know if $||\bar{r}_P||_\infty$ has already dropped below or still exceeds the tolerance θ.

Approach 2. This second approach is derived from the previous one. As long as the active set $\tilde{S}$ is not empty yet, a patch P is picked and removed from $\tilde{S}$. Again, we distinguish two cases.

1. If $||\bar{r}_P||_\infty \leq \theta$, the next patch is selected and removed from $\tilde{S}$, and the algorithm continues.

2. If $||\bar{r}_P||_\infty > \theta$, P is relaxed until $||\bar{r}_P||_\infty \leq \theta$ and, afterwards, only the neighbors of P are inserted into the active set $\tilde{S}$.

Approach 3. Our third approach uses a selective policy for activating neighboring patches in order to reduce computational work. It is based on an additional parameter $\delta = \delta(\theta)$, $0 < \delta < \theta$. The idea is to perform more work on the current patch than required by relaxing its equations until we have $||\bar{r}_P||_\infty \leq \theta - \delta$. As a consequence, the current patch will not necessarily be inserted into the active set $\tilde{S}$ again, if any of its neighboring patches are relaxed subsequently.

The algorithm thus works as follows. As long as the active set $\tilde{S}$ is not empty yet, a patch P is picked and removed from $\tilde{S}$. We again distinguish two cases.

1. If $||\bar{r}_P||_\infty \leq \theta - \delta$, the next patch is selected and removed from $\tilde{S}$, and the algorithm continues.

2. If $||\bar{r}_P||_\infty > \theta - \delta$, P is relaxed until $||\bar{r}_P||_\infty \leq \theta - \delta$ and, afterwards, any neighbor P' of P is inserted into the active set $\tilde{S}$, if it cannot be guaranteed that the absolute value of each of its scaled residuals θ_j, $j \in P'$, is less or equal than the tolerance θ.

Obviously, this third approach is more sophisticated than the first and the second one. Since the current patch P is relaxed until $||\bar{r}_P||_\infty$ falls below $\theta - \delta$, it will only be inserted into the active set $\tilde{S}$ again, if the relaxation of one of its neighboring patches causes the absolute value of at least one component θ_i of $\bar{r}_P$ to increase "significantly". In the following, we will describe this idea in more detail.

Assume that P' is a neighboring patch of P and that P' is currently not in the active set $\tilde{S}$. After the relaxation of P, we have to decide whether P' must be inserted into $\tilde{S}$ or not. For this purpose, it is sufficient to consider the changes of the scaled residuals θ_j at the respective boundary nodes $j \in P'$ because (due to the assumption of compact 9-point stencils) these are the only nodes whose scaled residuals are influenced by iterates u_i, $i \in P$. We must distinguish two cases, which will be discussed subsequently:

- *Case 1:* P' is the eastern, southern, western, or northern neighbor of P.

- *Case 2:* P' is the southeastern, northeastern, southwestern, or northwestern neighbor of P.

Case 1. For P' being the eastern neighbor of P, the situation is illustrated in Figure 8.3. Whether its neighboring patch P' will be inserted into the active set $\tilde{S}$ is determined by the changes $\delta\theta_j$ of the scaled residuals at the corresponding interior boundary nodes $j \in P'$ (the dark gray nodes in Figure 8.3) and at the corresponding corner nodes $j \in P'$ (the black nodes in Figure 8.3).

For each interior boundary node $j \in P'$, we can either calculate its scaled residual θ_j completely, or we can apply the result from Section 8.4.5.2 in order to determine the change $\delta\theta_j$ of its scaled residual. As is depicted in Figure 8.3 for one interior boundary node of P', the equation at each interior boundary node $j \in P'$ involves the values u_{i_1}, u_{i_2}, and u_{i_3} at three consecutive boundary nodes $i_1, i_2, i_3 \in P \cap \mathrm{Conn}(j)$. Hence, $\delta\theta_j$ can be calculated as

$$\delta\theta_j = \frac{a_{j,i_1}\delta_{i_1} + a_{j,i_2}\delta_{i_2} + a_{j,i_3}\delta_{i_3}}{a_{j,j}} \ ,$$

which is less expensive than determining the scaled residual θ_j completely. According to Section 8.4.5.2, the values δ_{i_1}, δ_{i_2}, and δ_{i_3} represent the changes of the approximations u_{i_1}, u_{i_2}, and u_{i_3} at the grid nodes i_1, i_2, and i_3, respectively.

In order that the respective changes δ_{i_1}, δ_{i_2}, and δ_{i_3} are available and the algorithm thus works correctly, it is essential that the approximations u_i along the corresponding boundary of patch P are saved to separate memory locations whenever the relaxation of P has caused its neighbor P' to be inserted into the active set $\tilde{S}$. After the relaxation of P, the changes δ_i are then given by the differences of the new iterates u_i along the corresponding boundary of P and the old iterates which have been saved previously.

For each node j located in one of the two corresponding corners of P' (the black nodes in Figure 8.3), the corresponding equation involves iterates at nodes belonging to four different patches. Since a series of small changes can add up to a significant change of the absolute value of the scaled residual θ_j, it is not sufficient to determine the change which results from the previous relaxation of patch P only. Instead, possible changes of all three neighboring patches must be accounted for and, hence, the scaled residual θ_j must be calculated explicitly.

Obviously, the situation is analogous if patch P' is the northern, the southern, or the western neighbor of patch P.

Case 2. For P' being the southeastern neighbor of P, the situation is illustrated in Figure 8.4. In this case, we explicitly have to calculate the scaled residual θ_j at the corresponding corner node $j \in P'$ (the black node in Figure 8.4). Since the equation at node j covers values located on four different patches, this case parallels the handling of the corner nodes in the previous case. Thus, the scaled residual θ_j must again be determined explicitly. Note that j is the only node of P' whose equation involves the approximation at a node belonging to patch P. Hence, the neighboring patch P' will only be inserted into the active set $\tilde{S}$, if the relaxation of P causes $|\theta_j|$ to exceed θ.

Obviously, the situation is analogous for P' being the northeastern, the southwestern, or the northwestern neighbor of P.

In the following, we will present an algorithmic formulation of the third relaxation/activation approach.

8.4.6 Algorithmic Description of the Patch-Adaptive Relaxation Scheme

Algorithm 8.4 parallels the point-oriented adaptive relaxation scheme. Both algorithms have in common that they terminate as soon as the active set $\tilde{S}$ is empty. This implies that, upon termination, we have $|\theta_i| \leq \theta$, $1 \leq i \leq n_l$; i.e., the absolute values of all scaled residuals θ_i on the corresponding level l have fallen below the prescribed tolerance θ.

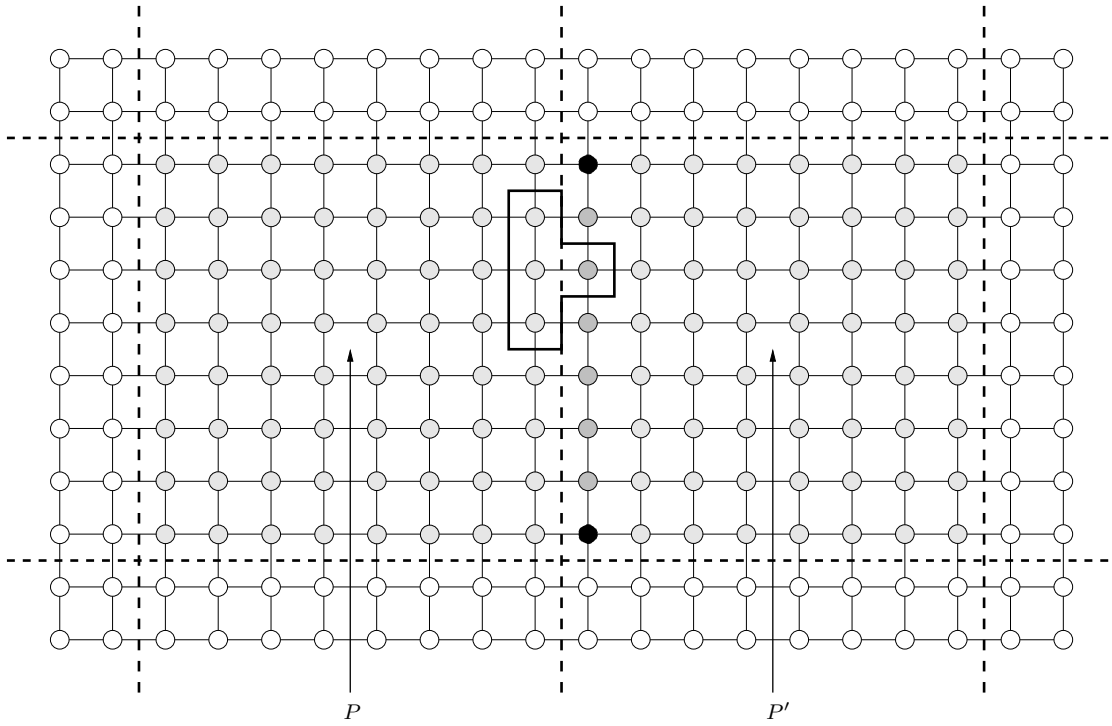


Figure 8.3: Activation of the eastern neighboring patch P' of P.

It is important to note that, despite this common property, the patch-based adaptive iteration must not be interpreted as a special version of the flexible point-based algorithmic framework presented in Section 8.2. Rather, the patch-based adaptive relaxation scheme represents a *variant* of the point-based adaptive relaxation scheme. As a consequence of the property that each patch is only treated as a whole, the patch-based method may cause equations to be relaxed, which would not be relaxed in the course of the point-based adaptive iteration.

From a theoretical viewpoint, it is the uniqueness of the solution of the minimization problem on each grid level l which allows that each patch $P \in \mathcal{P}_l$ is considered and treated as an indivisible entity, instead of selecting and relaxing individual equations $i \in P$ only. See again Sections 8.2.2 and 8.3.2

Moreover, the patch-based smoother does not reduce the efficiency of the subsequently performed relaxations on coarser grids. This property can be explained as follows. Whenever the

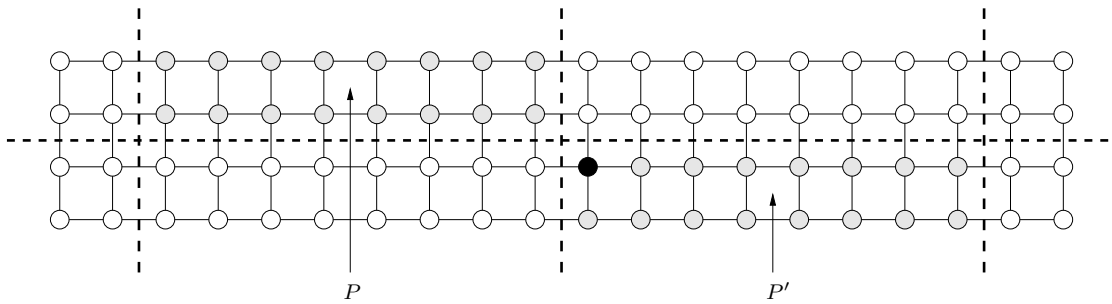


Figure 8.4: Activation of the southeastern neighboring patch P' of P.

Algorithm 8.4 Patch-adaptive relaxation on grid level l.

Require: tolerance $\theta = \theta_l > 0$, tolerance decrement $\delta = \delta(\theta)$, $0 < \delta < \theta$,
initial guess $u := u_l^{(0)} \in \mathbf{R}^{n_l}$, initial active set $\tilde{S} := \tilde{S}_l^{(0)} \subseteq \mathcal{P}_l$

```
1: while S̃ ≠ ∅ do
2:    Pick P ∈ S̃
3:    S̃ ← S̃ \ {P}
4:    if ||r̄_P||_∞ > θ − δ then
5:       relax(P)
6:       activateNeighbors(P)
7:    end if
8: end while
```

Ensure: $S_l = \emptyset$ // *The strictly (!) active set* S_l *is empty, see* (8.7)

patch-adaptive smoother has terminated on level l, all scaled residuals θ_i, $1 \leq i \leq n_l$, on level l are "small" (as defined by the corresponding tolerance θ_l). If, however, the algebraic error on level l is still relatively large, there will be relatively large scaled residuals on coarser levels. Consequently, continuing the (patch-adaptive) multigrid iteration on the next coarser level will maintain its numerical efficiency. Hence, according to the fundamental principle of the fully adaptive multigrid method, the objective of the patch-adaptive multigrid approach is to efficiently reduce the scaled residuals on all levels of the grid hierarchy, cf. Section 8.3.4.

Analogous to the case of the point-based adaptive relaxation scheme presented in Section 8.2.3, the patch-adaptive algorithm does not represent a linear iterative method either. This is due to the fact that the update order of the patches is not fixed and that, depending on the problem to be solved, some patches may be relaxed more frequently than others.

As is the case for the point-based adaptive relaxation scheme, the patch-based adaptive algorithm performs particularly well as a multigrid smoother as soon as the problem exhibits singularities or perturbations from boundary conditions. In such cases, the introduction of local relaxation can yield sufficiently smooth algebraic errors on the fine grids of the hierarchy, cf. Section 8.3.1. By construction, local relaxation is automatically accomplished by the adaptive processing approach. See the discussion of the model problem in Section 8.5 below.

Note that, for smooth problems, the introduction of patch adaptivity is commonly not worth the effort. Rather, the additional overhead based on the adaptive processing scheme may even cause an increase in execution time.

8.4.7 Remarks on Implementation Details

The patch-adaptive multigrid code which our subsequent experiments are based on has been implemented in ANSI C++. A comprehensive description of our object-oriented implementation is beyond the scope of this thesis. Instead, we refer to [Chr03] for details.

In our code, patches are represented as relatively small objects only. In order to simplify the implementation, each patch object stores the coordinates of the corner nodes; i.e., the position of its corresponding rectangular frame. The actual approximations u_l, the matrices A_l, as well as the right-hand sides f_l are stored as global array objects. Hereby, the matrices A_l are represented as arrays of (compact) stencils. Additionally, each patch object comprises four arrays of floating-point

numbers in order to save the values along its interior boundaries[4]. As was explained in detail in Section 8.4.5.3, this step is necessary to compute the changes of the scaled residuals at nodes belonging to neighboring patches.

Note that in the case of a parallel implementation of a patch-oriented multigrid scheme, the patches would be distributed among all available processing nodes and each patch would typically store its own part of the linear system of the corresponding grid level. We refer to [Löt96] for the discussion of such a parallel patch-based multigrid implementation.

In our patch-adaptive multigrid code, each active set $\tilde{S}_l$ is implemented as a first in, first out (FIFO) buffer. This choice has proven to yield reasonable results in all test cases. See [Dau01] for a discussion of alternative implementations of the active set for the case of a point-based adaptive relaxation scheme, including last in, first out (LIFO) buffers (stacks) as well as priority queues.

Whenever a patch $P \in \mathcal{P}_l$ is relaxed, the maximum norm $||\bar{r}_P||_\infty$ of the scaled patch residual $\bar{r}_P$ is only computed after every four patch relaxations in order to reduce the overhead due to residual computations. Hence, for any patch on any grid level, the number of patch relaxations will be a multiple of 4, cf. the results presented in Section 8.5.2.2. Note that this factor has been chosen empirically and proved reasonable in our tests. For different problems, however, more efficient adaptive multigrid schemes may result for different values.

In order to simplify the implementation, Dirichlet boundary nodes are stored explicitly. However, they are not considered as unknowns. Formally, the Dirichlet boundary values contribute to the right-hand sides f_l, $0 \leq l \leq L$, of the corresponding linear systems.

Our implementation of the Galerkin products follows the detailed algorithmic description in [Wes92]. However, for the sake of simplicity, we exploit the observation that, if a fine-grid operator A_l, $l > 0$, and the corresponding inter-grid transfer operators I_l^{l-1} and I_{l-1}^l are characterized by compact 9-point stencils, the resulting coarse-grid operator A_{l-1} will be characterized by compact 9-point stencils as well, cf. Section 3.3.5.3. This result permits the use of efficient cache-aware data structures on all grid levels. We have particularly implemented the access-oriented data layout from Section 5.3.2.

Our patch-adaptive multigrid scheme implements various relaxation/activation approaches. However, all experimental results we will present in the following section are based on Approach 3 from Section 8.4.5.3.

8.5 Experimental Results

The purpose of the following discussion is primarily to demonstrate the increased robustness and the resulting enhanced numerical efficiency of the patch-adaptive multigrid method when applied to an appropriate problem, which exhibits an operator-induced singularity, for example. In particular, we will focus on a standard model problem defined on a L-shaped domain which is characterized by a re-entrant corner singularity.

The inherent cache efficiency exhibited by an appropriate implementation of the patch-adaptive multigrid algorithm has been explained in Section 8.4.3.2. In this thesis, we have omitted the presentation of performance results (e.g., MFLOPS rates and memory hierarchy profiling data) since the tuning of our patch-adaptive multigrid code is the objective of current research activities in the DiME project. We refer to the presentation and the promising performance analysis of a patch-based multigrid implementation in [Löt96, LR97].

[4] In the case of a boundary patch, some of these arrays may be unused.

8.5.1 Description of the Model Problem

8.5.1.1 Definition

We define an L-shaped domain $\Omega \subset \mathbf{R}^2$ as follows:

$$\Omega := \{(x, y) \in (-0.5, 0.5)^2;\ x < 0 \text{ or } y > 0\} \ . \tag{8.23}$$

This domain is illustrated in Figure 8.5. We introduce polar coordinates (r, φ), $r \geq 0$, $0 \leq \varphi \leq 2\pi$, in 2D as usual; i.e., we have

$$\begin{aligned} x &= r\cos\varphi \ , \\ y &= r\sin\varphi \ . \end{aligned}$$

We consider the Dirichlet boundary value problem

$$-\Delta u = 0 \quad \text{in} \quad \Omega \ , \tag{8.24}$$

$$u = \sin\left(\frac{2}{3}\varphi\right) r^{\frac{2}{3}} \quad \text{on} \quad \partial\Omega \ , \tag{8.25}$$

where, as usual, $\partial\Omega$ denotes the boundary of Ω.

Note that, since $\sin 0 = \sin \pi = 0$, the boundary conditions particularly imply that the solution u vanishes on $\partial\Omega_1 \cup \partial\Omega_2 \subset \partial\Omega$, where the parts $\partial\Omega_1$ and $\partial\Omega_2$ of $\partial\Omega$ are defined as depicted in Figure 8.5. Similar boundary value problems with re-entrant corners are analyzed in [Bra97, Hac96].

8.5.1.2 Analytical Solution

According to [Hac96], the two-dimensional Laplace operator

$$\Delta := \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

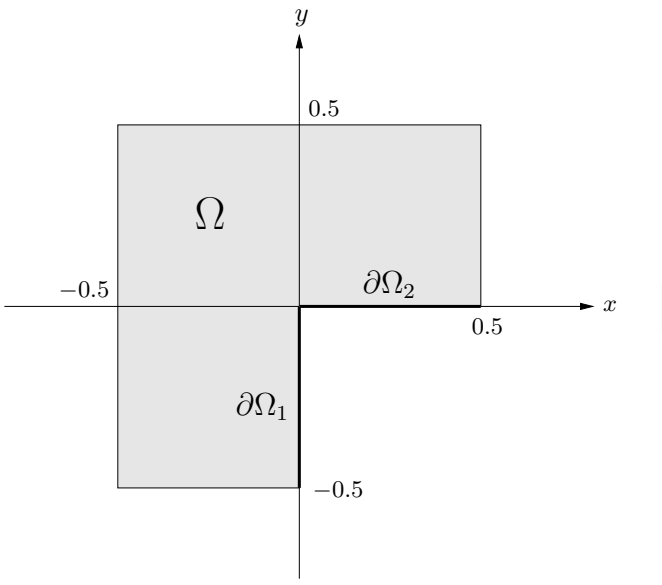


Figure 8.5: L-shaped domain Ω of the model problem.

can be represented in polar coordinates as

$$\Delta = \frac{\partial^2}{\partial r^2} + \frac{1}{r}\frac{\partial}{\partial r} + \frac{1}{r^2}\frac{\partial^2}{\partial \varphi^2} \ .$$

Using this representation of Δ in polar coordinates, it is straightforward to verify that

$$u(r,\varphi) := \sin\left(\frac{2}{3}\varphi\right) r^{\frac{2}{3}} \tag{8.26}$$

is a solution of (8.24) and further satisfies the Dirichlet boundary conditions given by (8.25).

Alternatively, the fact that u is harmonic (i.e., $\Delta u = 0$) can be verified by considering the complex function f, $f : \mathbf{C} \to \mathbf{C}$, with $f(z) = z^{\frac{2}{3}}$. Using the identity $z = re^{i\varphi}$, $r \geq 0$, $0 \leq \varphi < 2\pi$, we find that $u = \mathrm{Im} f(z)$. Since f is holomorphic (analytic) for $r > 0$, it follows that $\Delta u = 0$ for $r > 0$ [Ahl66].

Note that $u \in C^2(\Omega) \cap C^0(\bar{\Omega})$ holds. Hence, u is a classical solution of (8.24), (8.25), which can be shown to be unique [Bra97]. However, we have $u \notin C^1(\bar{\Omega})$. This singular behavior of u results from the fact that the first derivatives of u are not bounded in $r = 0$; in fact, we obtain for $r > 0$ that

$$\frac{\partial u(r,\varphi)}{\partial r} = \frac{2}{3}\sin\left(\frac{2}{3}\varphi\right) r^{-\frac{1}{3}} \ .$$

Consequently, for any fixed value of φ, $0 < \varphi < \frac{3}{2}\pi$, it follows that

$$\lim_{r \to 0} \frac{\partial u(r,\varphi)}{\partial r} = \infty \ .$$

Figure 8.6 illustrates the analytical solution u of (8.24), (8.25) using a regular grid of 65×65 nodes.

Note that it is the singular behavior of the solution u in $r = 0$ which generally spoils the numerical efficiency (i.e., the convergence behavior) of classical multigrid algorithms. Instead, it favors multigrid methods based on adaptive relaxation techniques, which automatically introduce local relaxation. See Section 8.5.2 below.

8.5.1.3 Finite Element Discretization

The general steps involved in the numerical solution of PDEs have already been outlined in Section 3.1. In our case, the model problem (8.24), (8.25) is discretized on an equidistant finite element mesh covering the L-shaped domain Ω. Our discretization approach is based on square elements and bilinear basis functions φ_i; i.e., polynomials of the form

$$\varphi_i(x,y) := a_{i,0} + a_{i,1}x + a_{i,2}y + a_{i,3}xy \ .$$

As usual, we use h to denote the mesh size. The computation of the element stiffness matrices, the subsequent assembly of the (global) stiffness matrix A_h, as well as the generation of the load vector f_h is straightforward, yet beyond the scope of this thesis. Instead, we refer to the finite element literature; e.g., [Bra97, JL01, KA00, Sch91].

In summary, the standard finite element approach yields a system of linear equations which is equivalent to

$$A_h u_h = f_h \ . \tag{8.27}$$

Here, $A_h := (a_{i,j})_{1 \leq i,j \leq n}$ represents the stiffness matrix, the unknown vector u_h contains the coefficients of the basis functions at the interior grid points, and the right-hand side f_h represents the influence of the Dirichlet boundary values at the grid points located on the boundary $\partial\Omega$.

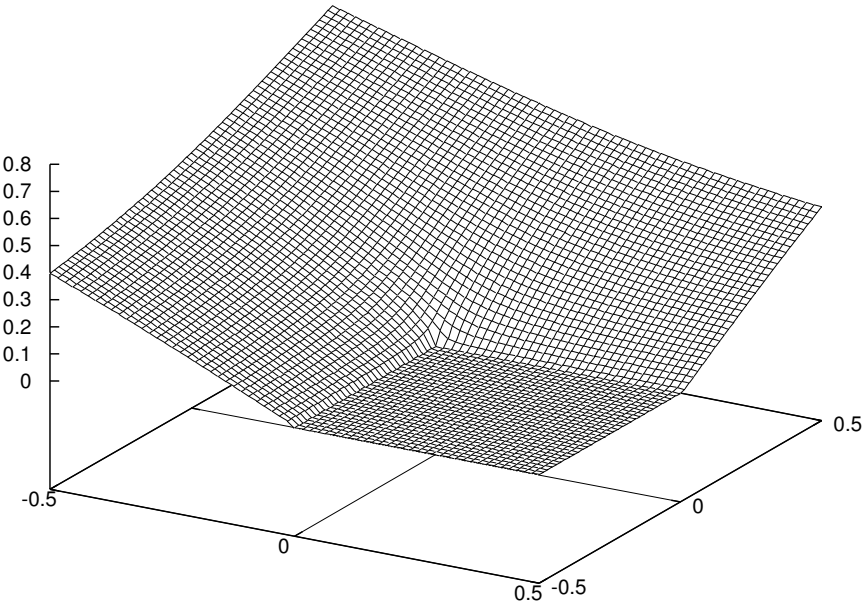


Figure 8.6: Exact solution u of the model problem given by (8.24) and (8.25) on the L-shaped domain Ω.

In particular, at each interior grid node, we obtain the constant stencil

$$\frac{1}{3}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \ , \tag{8.28}$$

see [BHM00, Rüd93b]. Obviously, the stiffness matrix A_h is symmetric. Furthermore, A_h is weakly diagonally dominant in the following sense: for all i, $1 \leq i \leq n$, we have

$$|a_{i,i}| \geq \sum_{j \neq i} |a_{i,j}| \ ,$$

where, in addition,

$$|a_{i',i'}| > \sum_{j \neq i'} |a_{i',j}|$$

holds for at least one index i', $1 \leq i' \leq n$. This follows from the constant entries of the stencil — see (8.28) — and from the usual treatment of the Dirichlet boundary conditions. Finally, A_h is irreducible; i.e., for any two indices i, j, $1 \leq i, j \leq n$, either $a_{i,j} \neq 0$ holds, or it exists a series $i_1, i_2, \ldots, i_s$ of indices, $1 \leq i_1, i_2, \ldots, i_s \leq n$, such that $a_{i,i_1} a_{i_1,i_2} \ldots a_{i_s,j} \neq 0$. These three properties (symmetry, weak diagonal dominance, and irreducibility) imply that A_h is symmetric positive definite; see [Sch97] and the references provided therein. As a consequence, the adaptive numerical schemes which have been presented previously can be applied to the linear system (8.27).

Note that the boundary value problem (8.24), (8.25) can be transformed into an equivalent problem with homogeneous Dirichlet boundary conditions. For this purpose, we choose a function $u_0 \in C^2(\Omega) \cap C(\bar{\Omega}) \cap H^1(\Omega)$ such that it coincides with the inhomogeneous boundary conditions on $\partial\Omega$; i.e., $u_0 = u$ on $\partial\Omega$, cf. (8.25) and the definition of u given by (8.26).

If we introduce the functions $\tilde{u} := u - u_0$ and $\tilde{f} := \Delta u_0$, (8.24), (8.25) can be rewritten as follows:

$$-\Delta\tilde{u} = \tilde{f} \quad \text{in} \quad \Omega \ , \tag{8.29}$$

$$\tilde{u} = 0 \quad \text{on} \quad \partial\Omega \ . \tag{8.30}$$

Note that this new formulation of the boundary value problem involves homogeneous Dirichlet boundary conditions only. Using integration by parts, (8.24) is then transformed into its weak form:

Find $\tilde{u} \in H_0^1(\Omega)$ such that for all test functions $v \in H_0^1(\Omega)$ the following relation holds:

$$a(\tilde{u}, v) = (\tilde{f}, v)_{L_2} \ . \tag{8.31}$$

For $\tilde{u}, v \in H_0^1(\Omega)$, the symmetric and positive bilinear form a, $a : H_0^1(\Omega) \times H_0^1(\Omega) \rightarrow \mathbf{R}$, is given by

$$a(\tilde{u}, v) := \int_\Omega \langle \nabla\tilde{u}, \nabla v \rangle dx \ ,$$

where $\langle \cdot, \cdot \rangle$ denotes the standard inner product in $\mathbf{R}^2$ (see also Section 4.2.2) and $(\cdot, \cdot)_{L_2}$ stands for the $L_2(\Omega)$ inner product:

$$(\tilde{u}, v)_{L_2} := \int_\Omega \tilde{u} v dx \ .$$

Subsequently, in order assemble the linear system (8.27), $\tilde{u}$ is approximated in the appropriate finite-dimensional space V_h defined by the bilinear basis functions φ_i; i.e., $V_h := \text{span}\{\varphi_i\}$. In our case, the test functions v are chosen from V_h as well. We refer to [Bra97] for details.

The linear system (8.27) represents the starting point for the application of our patch-adapted multigrid method from Section 8.4. In Section 8.5.2, we will present an experiment in order to demonstrate the numerical efficiency of our approach.

Note that the application of the point-based adaptive relaxation scheme from Section 8.2 to an equivalent model problem is discussed in [Rüd93b]. For this purpose, Rüde uses a hierarchy of adaptively refined triangular grids in order to improve the accuracy of the finite element discretization in the vicinity of the singularity.

8.5.2 Numerical Tests

8.5.2.1 Preparations

Partitioning of the L-shaped grid levels. In order that the patch-adaptive multigrid scheme can be applied, it is necessary that each grid level on which the patch-adaptive smoother is used is split into patches. This partitioning is demonstrated in Figure 8.7. For ease of presentation, Figure 8.7 shows the division of a relatively small square grid of 33×33 nodes into relatively small patches of approximately 8×8 nodes each. Using the notation from Section 3.3.2, Figure 8.7 refers to the case $n_{\text{dim}} = 32$.

Dirichlet boundary nodes are drawn as black points, whereas the grid points corresponding to unknowns are illustrated as dark gray nodes. In order that the geometry of the L-shaped domain is respected on each grid level, the solution values stored at the respective white nodes are never modified in the course of the multigrid iterations. They are initialized with 0 (on all grid levels), and the patches they belong to are never inserted into the active sets $\tilde{S}_l$. Consequently, on each level of the grid hierarchy, the white nodes located along the interior boundary can be considered as Dirichlet boundary nodes as well.

Note that the patch sizes are chosen asymmetrically such that all grid nodes i at positions (x_i, y_i) with $x_i \geq \frac{n_{\text{dim}}}{2}$ and $y_i \leq \frac{n_{\text{dim}}}{2}$ are located on patches whose approximations will never be modified. As a consequence, there are $(n_{\text{dim}} - 1)^2 - \left(\frac{n_{\text{dim}}}{2}\right)^2$ unknowns located on the corresponding level of the grid hierarchy.

The treatment of this model problem requires that our patch-adaptive multigrid implementation is adapted slightly. See [Chr03] for details.

Grid hierarchy and patch size. In the following experiment, we have used a hierarchy of ten grid levels. The mesh width on the finest level L is $h_L := \frac{1}{2048}$, and the mesh width on the coarsest level $l = 0$ is $h_0 := \frac{1}{4}$. The coarsest level $l = 0$ comprises five unknowns, as is illustrated in Figure 8.8.

We have chosen a level-independent patch size of approximately 32×32 grid nodes, cf. Section 8.4.2. The patch-adaptive smoother is used on all levels l with $h_l \leq \frac{1}{64}$; i.e., on the six finest levels of the grid hierarchy. Within each patch, we have assumed a lexicographic ordering of the unknowns. On each of the four coarsest levels, a standard Gauss-Seidel smoother based on a lexicographic ordering of the unknowns has been used.

Residual reduction factor. In our experiments, we have measured the efficiency of any individual iteration k, $k > 0$, by dividing the maximum norm of the scaled residual after the k-th iteration by its maximum norm before the k-th iteration:

$$\mu^{(k)} := \frac{||\bar{r}^{(k)}||_\infty}{||\bar{r}^{(k-1)}||_\infty} \ .$$

The value $\mu^{(k)}$ is called the *residual reduction factor* corresponding to iteration k. Therefore, the *asymptotic residual reduction factor* is given by

$$\mu^{(\infty)} := \lim_{k \to \infty} \mu^{(k)} \ .$$

Determination of residual reduction factors. The objective of our experiment is to demonstrate the robustness and the resulting numerical efficiency of the patch-adaptive multigrid approach. For this purpose, we have slightly modified the model problem from Section 8.5.1 by changing the boundary conditions. Instead of requiring the boundary conditions defined by (8.25), we impose homogeneous Dirichlet boundary conditions. Hence, the exact solution of the model problem is given by $u(r, \varphi) := 0$. Note that this change only influences the right-hand sides of the hierarchy of linear systems, while the discrete operators on all levels and, therefore, the convergence behavior of the multigrid scheme under consideration remain unchanged.

We use $u_L^{(0)} := (1, \ldots, 1)^T \in \mathbf{R}^n$ as initial guess on the finest level L of the grid hierarchy. Likewise, the choice of the initial guess does not influence the asymptotic convergence behavior of the iterative methods under consideration.

The advantage of this modified model problem is that the convergence behavior of iterative methods can be determined more accurately. The idea is to scale the numerical approximation after each iteration such that the (maximum) norm of the (scaled) residual is 1. The repeated scaling of the approximation ensures that the quality of the numerical results does not suffer from round-off errors and limited machine precision. This effect is particularly dramatic when examining multigrid methods which may converge very fast.

Note that this procedure corresponds to the application of the power method in order to determine an eigenvector to the eigenvalue of the iteration matrix M with the largest absolute value;

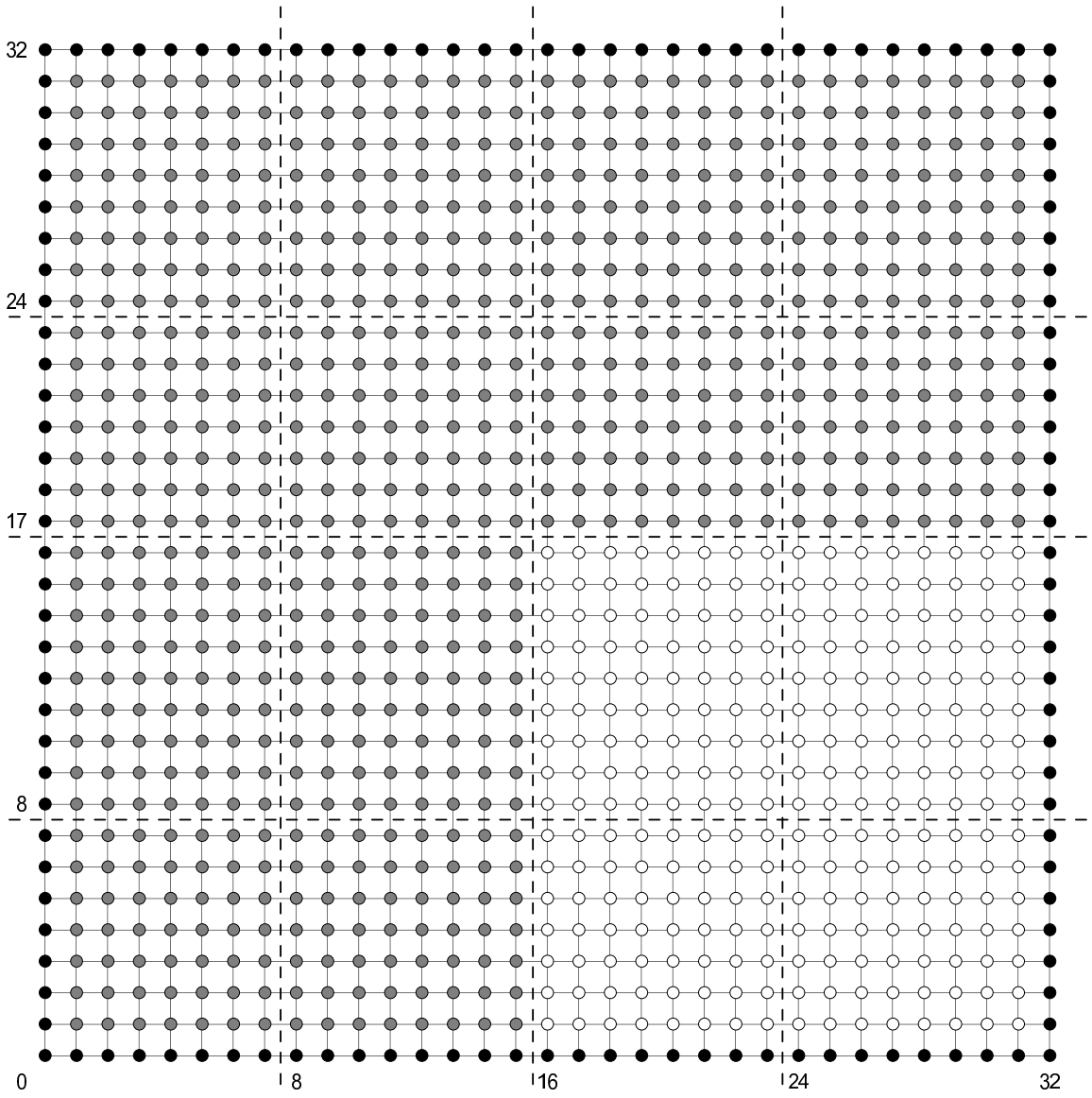


Figure 8.7: Partitioning of an L-shaped grid into patches, here: $h_l = \frac{1}{32}$.

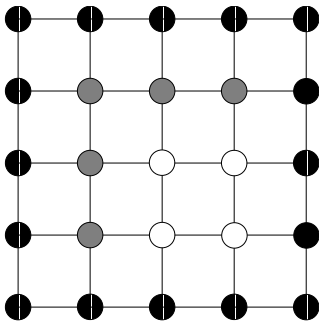


Figure 8.8: Discretization of the coarsest grid level on the L-shaped domain, $h_0 = \frac{1}{4}$.

i.e, to the dominant eigenvalue of M [BF01]. In other words, this procedure identifies an element of the eigenspace corresponding to the dominant eigenvalue of M. This vector therefore represents a component of the algebraic error which can be reduced least efficiently by the iterative scheme under consideration, cf. Section 3.2.1.3.

8.5.2.2 Results

Numerical efficiency of standard multigrid V(2,2)-cycles. For a smooth standard model problem which does not exhibit any singular behavior, the application of standard multigrid V(2,2)-cycles (based on a Gauss-Seidel smoother with a lexicographic ordering of the unknowns) has yielded an asymptotic residual reduction factor of $\mu^{(\infty)} \approx 0.04$. In contrast, the execution of such standard multigrid V(2,2)-cycles applied to the modified model problem on the L-shaped domain has revealed a significantly larger asymptotic residual reduction factor of $\mu_L^{(\infty)} \approx 0.08$.

This decrease in convergence speed confirms the observations which have led to the principle of local relaxation, see again Section 8.3.1. Note that this model problem only serves as a motivating example. In practice, even the "worse" convergence rate $\mu_L^{(\infty)}$ would still be considered excellent.

Residual reduction factors for standard multigrid V-cycle schemes have been presented in [Chr03] as well. However, the experiments discussed in [Chr03] are based on standard 5-point discretizations of the Laplacian using finite differences on each grid level. The observed residual reduction factors are significantly larger than the ones we have measured for the finite element discretization described in Section 8.5.1.3 and the definition of the coarse-grid matrices using Galerkin products. However, the respective ratios of the residual reduction factors for the smooth model problem and the model problem with the re-entrant corner singularity are comparable.

Configuration of the patch-adaptive multigrid V-cycle scheme. In order to demonstrate the robustness and the numerical efficiency of the patch-adaptive multigrid method, we have examined the numerical behavior of a single patch-adaptive multigrid V-cycle applied to the modified model problem on the L-shaped domain. For the sake of simplicity, only post-smoothing has been performed.

We have chosen a dominant eigenvector of the standard multigrid V(2,2)-cycle iteration matrix as initial guess. Therefore, this initial guess represents a "worst-case" algebraic error for the standard multigrid V(2,2)-cycle scheme. It has been obtained by executing a sufficiently large number of standard multigrid V(2,2)-cycles on the modified model problem, starting with an arbitrary initial guess and scaling the approximations repeatedly to avoid problems due to numerical round-off, cf. Section 8.5.2.1.

We have further prescribed a residual reduction factor of 0.04 on each level of the grid hierarchy on which the patch-adaptive smoother is employed. This means that we have required that the patch-adaptive smoother reduces the maximum norm $||\bar{r}_l||$ of the corresponding scaled residual by a factor of 0.04. We have therefore defined $\theta_l := 0.04 \cdot ||\bar{r}_l^{(0)}||$, where $\bar{r}_l^{(0)}$ denotes the initial scaled residual on level l; i.e., before the execution of the patch-adaptive multigrid V-cycle. The choice of the residual reduction factor has been guided by examinations of the behavior of standard multigrid V(2,2)-cycles applied to smooth standard model problems, see above.

In addition, we have empirically chosen $\delta_l := 0.2 \cdot \theta_l$, cf. Sections 8.4.5.3 and 8.4.6. Admittedly, the optimal selection of the parameters θ_l and δ_l for the respective levels of the grid hierarchy is not yet fully understood and provides room for further optimization.

Behavior of the patch-adaptive multigrid V-cycle scheme. The analysis of the behavior of the patch-adaptive multigrid V-cycle scheme confirms our expectations. On the finer grids (i.e., on those grids where the patch-adaptive smoother is employed), the patches located close to the singularity are relaxed more often than the others. This behavior is illustrated in Figure 8.9.

It can be observed that the patches on the coarsest grid level on which the patch-adaptive smoother is employed ($h_l = \frac{1}{64}$) are relaxed 32, 36, and 44 times, respectively. This means that the corresponding linear system is solved rather accurately. As a consequence, only the respective three patches located next to the singularity are relaxed on all finer levels of the grid hierarchy.

Since the ten grid levels contain a total of 4,186,126 unknowns, a standard multigrid V(2,2)-cycle requires $4 \cdot 4,186,126 = 16,744,504$ elementary update steps and exhibits an asymptotic residual reduction factor of $\mu_L^{(\infty)} \approx 0.08$, see above. In contrast, our patch-adaptive multigrid scheme with an enforced residual reduction factor of 0.04 requires 385,860 elementary update steps only. This demonstrates the increased numerical efficiency which the patch-adaptive multigrid scheme exhibits when applied to a suitable problem. Similar promising results have been presented in [Chr03].

Note that we have yet not counted the calculations of the scaled patch residuals, cf. Section 8.4.4.1. Even if a patch is not relaxed, its scaled patch residual must be determined at least once. This means that, additionally, the work corresponding to approximately $4.2 \cdot 10^6$ elementary update steps is performed. Yet, the overall computational work for the patch-adaptive multigrid V-cycle is still less than 30% of the overall computational complexity of a standard multigrid V(2,2)-cycle, cf. again [Chr03].

The previous observation again reflects the trade-off between computational work and execution speed, see Section 8.4.3.2. If smaller patches are chosen, the total number of elementary relaxation steps will be reduced. However, the overhead of maintaining the active sets $\tilde{S}_l$ will be increased, thereby reducing the node update rate.

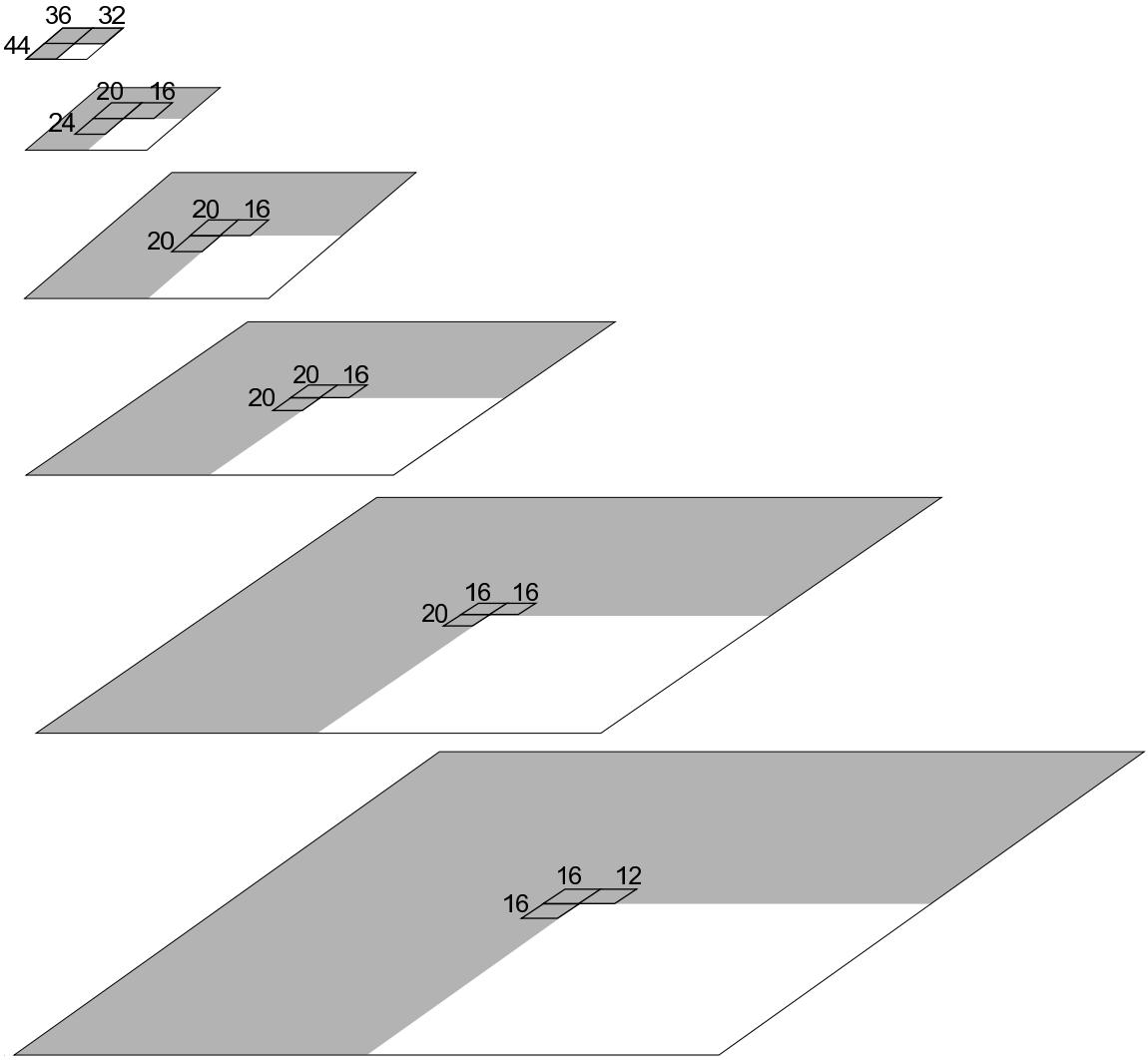


Figure 8.9: Numbers of patch relaxation on the six finest grid levels.

Part IV

Related Work, Conclusions, and Future Work

Chapter 9

Related Work — Selected Topics

9.1 Optimization Techniques for Numerical Software

9.1.1 Overview

Numerous references to related work have already been provided in the previous chapters. In addition, we refer to the detailed overview of related research activities presented in Weiß' thesis [Wei01]. [Wei01, §7] particularly contains a comprehensive discussion of profiling tools and software frameworks to simulate and visualize the cache behavior of application codes, see also Section 2.5.2.

In the following, we will give an overview of additional related research efforts covering both practical as well as theoretical aspects of data locality optimizations.

9.1.2 Automatic Performance Tuning of Numerical Software

Typically, due to the enormous complexity of real computing platforms, analytical techniques to determine optimization parameters (e.g., array paddings and block sizes) are based on simplifying assumptions concerning the properties of the underlying computer architectures. In many cases, however, the resulting machine models are not precise enough. For example, most abstract cache models assume the inclusion property and do not cover exclusive caches, as they are employed in several current AMD microprocessors, cf. Section 2.2.2. As a consequence, these analytical techniques often yield only suboptimal configurations of optimization parameters.

Furthermore, the current pace of hardware and software evolution renders the manual optimization of numerically intensive codes more difficult. Due to its complexity, this optimization process might even exceed the lifetime of the underlying computing platform which includes the hardware itself, but also the operating system and the compilers.

The two previous observations have led to the development of *automatic performance tuning* techniques and to the principle of *self-tuning* numerical codes. In order to refer to this pragmatic approach towards high performance software generation by successively evaluating the efficiency of alternative implementations, Dongarra et al. have introduced the notion of the *AEOS (automated empirical optimization of software)* paradigm [WD98, WPD01].

A widely used software package that is based on the AEOS paradigm is *ATLAS (Automatically Tuned Linear Algebra Software)* [WD98, WPD01][1]. The ATLAS project concentrates on the automatic application of empirical code optimization techniques for the generation of highly optimized

[1] See also `http://www.netlib.org/atlas`.

platform-specific BLAS libraries, see Section 9.2 for details. The basic idea is to successively introduce source-to-source transformations and evaluate the resulting performance, thus generating the most efficient implementation of the BLAS on the platform under consideration. Note that ATLAS still depends on an optimizing compiler for applying architecture-dependent optimizations and generating efficient machine code.

As was mentioned in Section 7.1.1, the generation of our cache-optimized multigrid codes and LBM implementations follows the AEOS paradigm as well. Similar techniques have further been applied in order to obtain highly tuned matrix multiply codes [BACD97] and optimized FFT implementations. For the latter, we refer to the *FFTW (Fastest Fourier Transform in the West)* research project [FJ98]. The related *SPIRAL* project strives for the automatic generation of platform-specific implementations of signal-processing algorithms such as the FFT and the discrete wavelet transform [PSX+04]. The research project *SPARSITY* targets the empirical platform-specific optimization of sparse matrix-vector multiplications [IY01], which form the basis of many numerical algorithms; e.g., the Krylov subspace methods mentioned in Section 3.2.1.3.

9.1.3 Further Optimization Techniques for Grid-Based Algorithms

9.1.3.1 Cache Optimizations for Unstructured Grid Multigrid

Unstructured grids. The cache performance optimizations on which our research has primarily focused target numerical computations based on highly regular grid structures. However, in order to handle complicated geometries, unstructured finite element meshes are employed by numerous applications in science and engineering, see [Bra97, JL01, KA00, Sch91].

As was mentioned in Section 6.4.1 in the context of loop blocking, the use of unstructured grids introduces a significant amount of overhead into the numerical simulation process. The resulting system matrices exhibit irregular sparsity patterns and, therefore, regular data structures such as the layouts from Section 5.3.2 cannot be used anymore. Instead, sparse matrix storage formats are typically employed; e.g., *compressed row storage (CRS)* or *jagged diagonal storage (JDS)* [BBC+94][2]. These data layouts have in common that they only store the nonzeros of the system matrices and thus need to maintain additional information about their row and column indices.

As a consequence, due to the indirect indexing of the matrix entries and the respective vector components, numerical computations involving such data structures require a larger portion of memory traffic than their counterparts based on regular data layouts. Moreover, the introduction of irregular data structures generally renders compiler-based code optimizations more difficult since less information about the data access patterns is available at compile time. These two aspects cause a considerable loss of performance.

Note that the increase in memory traffic implies a decrease in computational intensity; i.e., the ratio of floating-point operations to data access operations is reduced. This ratio, however, is already low in the case of multigrid methods since these schemes are characterized by an asymptotically linear computational complexity and thus exhibit high algorithmic efficiency, cf. Section 1.2.2. Consequently, due to the increased number of data accesses, implementations of multigrid methods on unstructured grids are even more bandwidth-limited than multigrid codes involving regular data structures [GKKS00].

Our joint research with Douglas et al. has striven for the optimization of the cache performance of multigrid algorithms on unstructured meshes in 2D and 3D. These multigrid algorithms are based

[2] Alternative to these algebraically oriented data structures (i.e., matrices and vectors), geometrically oriented irregular data structures representing the computational grid (i.e., elements, vertices, faces, edges, etc.) can be used.

on cache-efficient Gauss-Seidel smothers. In the following, we will briefly present the underlying idea. For detailed descriptions, we refer to our joint publications [DHK+00a, DHK+00b]. The most comprehensive presentation of these techniques for both the 2D and the 3D case can be found in Hu's PhD thesis [Hu00]. For an overview of our work, see also [DHH+00a, DHH+00b].

Preprocessing phase. The fundamental principle of these algorithms is that, in the course of a preprocessing (setup) phase, each level of the hierarchy is partitioned into contiguous blocks of grid nodes; the so-called *cache blocks*. This can be accomplished using the graph partitioning tool *METIS*[3]. In order to obtain efficient code execution, the sizes of the cache blocks should be adapted to the capacity of the cache level for which the application is tailored; i.e., the corresponding matrix rows, iterates, and components of the right-hand side should fit into this cache level simultaneously.

Within each cache block, sets of nodes which are located at given distances from the cache block boundary are identified. Afterwards, the nodes within each cache block are renumbered according to their distances from the cache block boundary. This requires the system matrices and the matrices representing the inter-grid transfer operators to be reordered correspondingly. This setup step is driven by the fundamental observation that, without referencing data from neighboring cache blocks, the grid nodes located in the interior of a cache block can be updated more often than the nodes located close to the cache block boundary.

Processing phase. The cache-efficient Gauss-Seidel smoother which is employed during the subsequent processing (iteration) phase visits the cache blocks of the current grid level one after the other and performs as many updates as possible on each cache block without referencing data from neighboring cache blocks. Consequently, this strategy requires that the cache blocks are revisited later on in order to become updated completely. In addition, the residual calculation is integrated into the update process of the cache blocks such that unnecessary sweeps over the respective grids are avoided.

9.1.3.2 Hierarchical Hybrid Grids

The *hierarchical hybrid grid (HHG)* framework represents an approach towards the simultaneous handling of unstructured and structured meshes in a parallel multigrid setting [HBR03, HKRG02]. The HHG principle comprises two steps. At first, the PDE to be solved is discretized using an unstructured finite element mesh. Then, in order to enhance the accuracy of the discretization, the individual elements are refined recursively in a highly structured fashion. Hence, this technique leads to *globally unstructured, locally structured* grids. This situation is illustrated in Figure 9.1, which is taken from [HBR03].

The HHG approach combines two major advantages. On one hand, it preserves the geometric flexibility arising from the discretization of the (potentially complicated) computational domains using finite elements. On the other hand, it uses patch-wise regularity and thus overcomes the performance penalty which is common for computations on irregular data structures, cf. Section 9.1.3.1 above. In particular, the regular data access patterns used within the individual elements of the unstructured coarsest grid can be exploited to achieve (cache-)efficient code execution.

Analogously, research on block-structured grids, overlapping grids, and structured adaptive mesh refinement (SAMR) is motivated by the objective to combine geometric flexibility and efficient code execution. See [BCGN00, BHQ98], for example.

[3]See the documentation under `http://www-users.cs.umn.edu/~karypis/metis`.

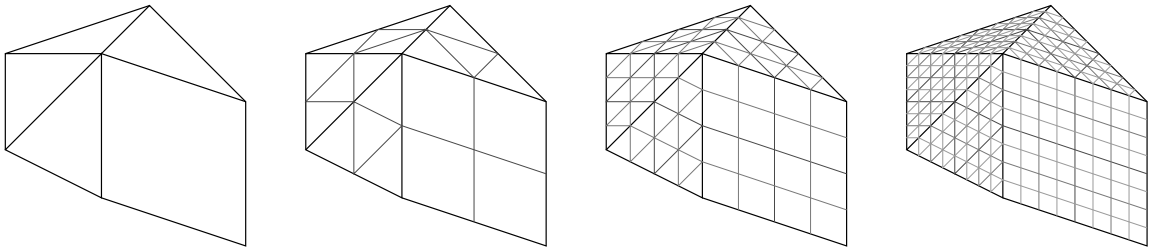


Figure 9.1: Locally regular refinement of a globally unstructured coarse grid.

9.1.3.3 Cache-Efficient Parallel Domain Decomposition Approaches

For the efficient parallel solution of time-dependent nonlinear PDEs, Keyes et al. have employed the class of *Newton-Krylov-Schwarz (NKS)* algorithms [Key02]. These numerical schemes follow the general design principle that, in each time step, an outer iterative Newton method [Sto99] is applied in order to linearize the nonlinear operator resulting from the discretization of the underlying nonlinear PDE or system of nonlinear PDEs. The individual linear systems occurring in the course of this Newton process are solved using a Krylov subspace method (e.g., GMRES), cf. Section 3.2.1.3.

In order to reduce the required number of iterations of the Krylov loop and the memory consumption of the Krylov subspace, appropriate preconditioners are employed [Gre97]. For this purpose, Keyes et al. use a domain decomposition technique; e.g., the additive Schwarz method [QV99]. Commonly, iterative algorithms such as multigrid V-cycles (cf. Section 3.3) or schemes based on (block-)incomplete LU factorizations of the corresponding system matrices [GL98] are then used as solvers within the individual subdomains. The domain decomposition method may further be equipped with a coarse-grid correction step in order to improve its overall convergence [Key02].

Keyes et al. have examined and optimized the performance of a parallel NKS implementation for the solution of problems in computational fluid dynamics on unstructured grids in 3D using large-scale machines. In particular, they have demonstrated that the single-node performance of the code can be enhanced significantly by applying suitable cache-based transformations, especially to the subdomain solver part [GKKS00]. These techniques cover the tuning of the data layouts (cf. Chapter 5) as well as the introduction of edge and node reordering, amongst others. We refer to the overview of potential optimization approaches given in [Key00].

Similar to the previous approach, Turek et al. have introduced a domain decomposition technique for the efficient parallel solution of problems in computational fluid dynamics [BKT99]. The principle of the researchers' *SCARC (Scalable Recursive Clustering)* approach is the decomposition of the computational domain into patches which can be processed in parallel. Within each patch, a highly regular mesh is used and cache-optimized *sparse banded BLAS (SBBLAS)* routines, which exploit the regular structure of the local matrices, can thus be employed to implement efficient subdomain solvers.

Multigrid schemes are then used in order to solve the global linear systems. In order to guarantee robustness and therefore numerical efficiency (in terms of fast global convergence), the local linear systems corresponding to the individual subdomains (patches) are solved using appropriate algorithms that respect the local grid structures, typically multigrid methods as well. Note that the local grids can be highly anisotropic for some of the patches. We refer to [Kil02] for a comprehensive presentation of the SCARC approach.

9.1.3.4 A Cache-Aware Parallel CA Implementation

In [DAC00], the authors present an approach to enhance the cache performance of parallel CA codes. They particularly focus on lattice-gas CA (see Section 4.2.1) and employ the graph partitioning tool METIS, cf. Section 9.1.3.1.

Firstly, the complete lattice is partitioned into subdomains, which are then assigned to the individual processors in the parallel computing environment. Secondly, each process uses METIS again in order to partition the subdomains into regions whose sizes are chosen according to the cache capacity. This approach thus parallels the techniques for cache-aware multigrid smoothers for unstructured grids presented in Section 9.1.3.1.

9.2 Cache Optimizations for the BLAS and LAPACK

9.2.1 Overview of the BLAS and LAPACK

Many numerical applications are based on elementary kernel routines which are provided by highly optimized underlying software libraries. In the case of algorithms involving dense matrices, the *BLAS (Basic Linear Algebra Subprograms)* as well as the *LAPACK (Linear Algebra PACKage)* libraries are often employed[4]. This section contains a brief description of these two libraries and presents optimization approaches in order to speed up the execution of their routines.

The BLAS library provides building blocks for performing elementary vector and matrix operations [DCHD90, Ueb97b]. In the following, we will use α and β to represent scalar values, whereas x and y denote vectors, and A, B, and C represent matrices.

The BLAS library is divided into three levels. The Level 1 BLAS do vector-vector operations; e.g., so-called $AXPY$ computations such as $y \leftarrow \alpha x + y$ and dot products such as $\alpha \leftarrow \beta + x^T y$. The Level 2 BLAS do matrix-vector operations; e.g., $y \leftarrow \alpha \mathrm{op}(A)x + \beta y$, where $\mathrm{op}(A) = A, A^T$, or A^H. Lastly, the Level 3 BLAS do matrix-matrix operations such as $C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$. Dedicated routines are provided for special cases such as symmetric and Hermitian matrices. The BLAS provide similar functionality for real and complex data types, in both single and double precision.

LAPACK represents another software package which is often integrated into numerical applications [ABB+99, Ueb97b]. LAPACK is based on the BLAS and implements routines for solving systems of linear equations, computing least-squares solutions of linear systems, and solving eigenvalue as well as singular value problems. The associated routines for factorizing matrices are also provided; e.g., LU, Cholesky, and QR decomposition. LAPACK can handle dense and banded matrices. In analogy to the BLAS library, LAPACK implements similar functionality for real and complex matrices, in both single and double precision.

9.2.2 Enhancing the Cache Performance of the BLAS

The presentation in this section closely follows the description of the ATLAS project provided in [WPD01], see also Section 9.1. ATLAS mainly targets the optimizations of the Level 2 and the Level 3 BLAS while relying on the underlying compiler to generate efficient Level 1 BLAS. This is due to the fact that the Level 1 BLAS basically contain no memory reuse and high-level source code transformations only yield marginal speedups.

[4] See also `http://www.netlib.org/blas` and `http://www.netlib.org/lapack`.

On the contrary, the potential for data reuse is high in the Level 2 and even higher in the Level 3 BLAS due to the occurrence of at least one matrix operand. Concerning the optimization of the Level 2 BLAS, ATLAS implements both register blocking[5] and loop blocking. In order to illustrate the application of these techniques it is sufficient to consider the update operation $y \leftarrow Ax + y$, where A is an $n \times n$ matrix and x, y are vectors of length n. This operation can also be written as

$$y_i \leftarrow \sum_{j=1}^{n} A_{i,j} x_j + y_i, \quad 1 \leq i \leq n \ ,$$

see [WPD01]. By keeping the current value y_i in a CPU register (i.e., by applying register blocking), the number of read/write accesses to y can be reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Furthermore, unrolling the outermost loop and hence updating k components of the vector y simultaneously can reduce the number of accesses to x by a factor of $\frac{1}{k}$ to $\frac{n^2}{k}$. This is due to the fact that each x_j contributes to each y_i. In addition, loop blocking can be introduced in order to reduce the number of main memory accesses to the components of the vector x from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ [WPD01]. This means that loop blocking can be applied in order to load x only once into the cache.

While the Level 2 BLAS routines require $\mathcal{O}(n^2)$ data accesses in order to perform $\mathcal{O}(n^2)$ floating-point operations, the Level 3 BLAS routines need $\mathcal{O}(n^2)$ data accesses to execute $\mathcal{O}(n^3)$ floating-point operations, thus exhibiting a higher potential for data reuse. Consequently, the most significant speedups are obtained by tuning the cache performance of the Level 3 BLAS; particularly the matrix multiply. This is achieved by implementing an L1 cache-contained matrix multiply and partitioning the original problem into subproblems which can be computed in cache [WPD01]. In other words, the optimized code results from blocking each of the three loops of a standard matrix multiply algorithm and calling the L1 cache-contained matrix multiply code from within the innermost loop. Figure 9.2 illustrates the blocked algorithm. In order to compute the shaded block of the product C, the corresponding blocks of its factors A and B have to be multiplied and added.

In order to further enhance the cache performance of the matrix multiply routine, ATLAS introduces additional blocking for either the L2 or the L3 cache. This is achieved by tiling the loop which moves the current matrix blocks horizontally through the first factor A and vertically through the second factor B, respectively. The resulting performance gains depend on various parameters; e.g., hardware characteristics, operating system features, and compiler capabilities [WPD01].

Note that fast matrix multiply algorithms which require $\mathcal{O}(n^\zeta)$, $\zeta < 3$, floating-point operations have been developed; e.g., Winograd's method and Strassen's method [Hig02]. These algorithms

[5]The developers of ATLAS refer to the term *register blocking* as a technique to explicitly enforce the reuse of CPU registers by introducing temporary variables.

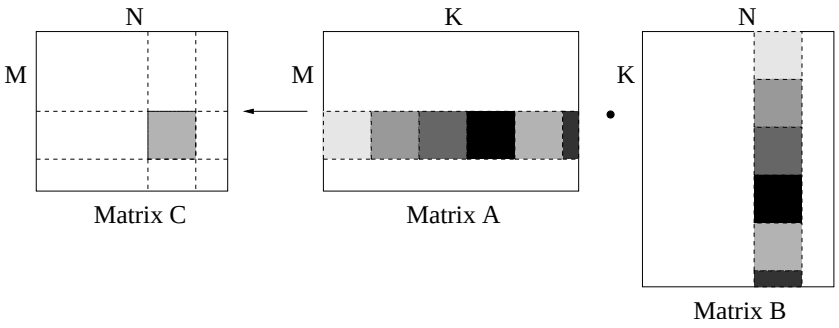


Figure 9.2: Blocked matrix multiply algorithm.

are based on the idea of recursively partitioning the factors into blocks and reusing intermediate results. However, error analysis reveals that these fast algorithms have different properties in terms of numerical stability. See [Hig02] for a detailed analysis.

9.2.3 Block Algorithms in LAPACK

In order to leverage the speedups which are obtained by optimizing the cache utilization of the Level 3 BLAS, LAPACK provides implementations of *block algorithms* in addition to the standard versions of various routines only based on the Level 1 and the Level 2 BLAS. For example, LAPACK implements block LU, block Cholesky, and block QR factorizations [ABB+99, Ueb97b]. The idea behind these algorithms is to split the original matrices into submatrices (blocks) and process them using highly efficient Level 3 BLAS routines, cf. Section 9.2.2.

In order to illustrate the design of block algorithms in LAPACK, we compare the standard LU factorization of a nonsingular $n \times n$ matrix A to the corresponding block LU factorization. In order to simplify the presentation, we initially leave pivoting issues aside. Each of these algorithms determines a lower unit triangular $n \times n$ matrix[6] L and an upper triangular $n \times n$ matrix U such that $A = LU$. The idea of this (unique) factorization is that any linear system $Ax = b$ can then be solved easily by first solving $Ly = b$ using a forward substitution step and subsequently solving $Ux = y$ using a backward substitution step [GL98, Hig02].

Computing the triangular matrices L and U essentially corresponds to performing Gaussian elimination on A in order to obtain an upper triangular matrix. In the course of this computation, all elimination factors $l_{i,j}$ are stored. These factors $l_{i,j}$ become the subdiagonal entries of the unit triangular matrix L, while the resulting upper triangular matrix defines the factor U. This elimination process is mainly based on the Level 2 BLAS; it repeatedly requires rows of A to be added to multiples of different rows of A.

The block LU algorithm works as follows. The matrix A is partitioned into four submatrices $A_{1,1}$, $A_{1,2}$, $A_{2,1}$, and $A_{2,2}$. The factorization $A = LU$ can then be written as

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} = \begin{bmatrix} L_{1,1} & 0 \\ L_{2,1} & L_{2,2} \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} \\ 0 & U_{2,2} \end{bmatrix} ,$$

where the corresponding blocks are equally sized, and $A_{1,1}$, $L_{1,1}$, and $U_{1,1}$ are square submatrices. Hence, we obtain the following equations:

$$A_{1,1} = L_{1,1}U_{1,1} , \tag{9.1}$$

$$A_{1,2} = L_{1,1}U_{1,2} , \tag{9.2}$$

$$A_{2,1} = L_{2,1}U_{1,1} , \tag{9.3}$$

$$A_{2,2} = L_{2,1}U_{1,2} + L_{2,2}U_{2,2} . \tag{9.4}$$

According to (9.1), $L_{1,1}$ and $U_{1,1}$ are computed using the standard LU factorization routine. Afterwards, $U_{1,2}$ and $L_{2,1}$ are determined from (9.2) and (9.3), respectively, using Level 3 BLAS solvers for triangular systems. Eventually, $L_{2,2}$ and $U_{2,2}$ are computed as the result of recursively applying the block LU decomposition routine to $\tilde{A}_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$. This final step follows immediately from (9.4). The computation of $\tilde{A}$ can again be accomplished by leveraging the Level 3 BLAS.

It is important to point out that the block algorithm can yield different numerical results than the standard version as soon as pivoting is introduced; i.e., as soon as a decomposition $PA = LU$ is

[6] A *unit* triangular matrix is characterized by having only 1's on its main diagonal.

computed, where P denotes a suitable permutation matrix [GL98]. While the search for appropriate pivots may cover the whole matrix A in the case of the standard algorithm, the block algorithm restricts this search to the current block $A_{1,1}$ to be decomposed into triangular factors. The choice of different pivots during the decomposition process may lead to different round-off behavior due to finite precision arithmetic.

Further cache performance optimizations for LAPACK have been developed. The application of recursively packed matrix storage formats is an example of how to combine both data layout as well as data access optimizations [AGGW02]. A memory-efficient LU decomposition algorithm with partial pivoting is presented in [Tol97]. It is based on recursively partitioning the input matrix. A formal framework for the derivation of efficient implementations of algorithms of numerical linear algebra — the *Formal Linear Algebra Methods Environment (FLAME)* — is described in [GGHvdG01]. In particular, [GGHvdG01] contains a historic overview of the development of efficient numerical software libraries.

9.3 The DiMEPACK Library

9.3.1 Functionality of the DiMEPACK Library

DiMEPACK is a library of cache-aware multigrid routines in 2D which is based on the data locality optimization techniques presented in Weiß' PhD thesis [Wei01] as well as in our previous chapters Chapters 5 and 6. Descriptions of DiMEPACK can also be found in [Wei01] and in more compact form in [KW01, KKRW01].

DiMEPACK implements both multigrid V-cycles and FMG cycles based on a CGC scheme. Full-weighting as well as half-weighting are implemented as restriction operators. The prolongation of the coarse-grid corrections is done using bilinear interpolation. DiMEPACK uses a Gauss-Seidel/SOR smoother based on a red-black ordering of the unknowns. See Section 3.3 for an introduction to multigrid methods and further references.

DiMEPACK can handle constant-coefficient problems based on discretizations using 5-point or 9-point stencils. It is applicable to problems on rectangular domains where different mesh widths in both space dimensions are permitted. It can handle both Dirichlet and Neumann boundary conditions [Hac96]. Equations for Neumann boundary nodes are obtained by introducing second-order central difference formulae for approximating the external normal derivatives. The arrays containing the boundary data have to be passed as parameters to the DiMEPACK library functions.

The user may specify the total number of grid levels. The linear systems on the coarsest grid are solved using LAPACK library routines [ABB+99]. For this purpose, the corresponding system matrix is split into two triangular factors in the beginning of the computation. If it is symmetric and positive definite, a Cholesky factorization is computed. Otherwise, an LU factorization is determined. In the course of the multigrid iteration, the coarsest systems are solved using forward-backward substitution steps. We refer to Section 3.2.1.2 for a brief overview of direct solvers for linear systems.

We have implemented various stopping criteria for the multigrid iterations which may be specified by the user. In general, the computation terminates as soon as either a maximal number of multigrid cycles has been performed or as soon as the discrete L2 norm or the maximum norm of the residual vector has dropped below a prescribed threshold.

Before the compilation of the DiMEPACK library, the user may set an environment variable in order to specify either single precision or double precision as the type of floating-point representation

to be used. According to the international standard IEC 559:1989 for floating-point numbers [Ueb97a], a single precision floating-point number occupies 4 B, whereas a double precision floating-point number occupies 8 B, cf. Section 7.3.2.4. As a consequence, the use of single precision floating-point numbers reduces memory traffic and roughly doubles the number of floating-point numbers which can be kept in cache. Hence, the choice of single precision arithmetic can significantly speed up code execution and can be interpreted as a cache optimization technique as well. Clearly, replacing double precision values by single precision values may have a significant impact on the quality of the numerical results [Gol91].

DiMEPACK can handle certain problem classes yielding singular matrices; e.g., Poisson problems with pure Neumann boundary conditions. In this case, individual solutions of the coarsest-grid problems are selected by arbitrarily choosing the value of the unknown in a corner of the computational domain [BP94]. We refer to the *DiMEPACK User Manual* [KKRW01] for a more detailed description of its functionality and features.

9.3.2 Optimization Techniques

9.3.2.1 Arithmetic Optimizations

Arithmetic optimizations do not aim at enhancing the cache performance, but at minimizing the number of floating-point operations to be executed without losing the identity of the numerical results[7]. The following arithmetic optimizations are implemented in DiMEPACK.

- DiMEPACK respects that, if both a 5-point stencil and a Gauss-Seidel smoother are used, the residuals vanish at the black grid nodes. This drastically simplifies the implementation of the restriction operators.
- In order to save multiply and divide operations, both the relaxation parameter of the smoothing routine (if different from 1) and the diagonal entries of the matrices are accounted for during a preprocessing step for the finest grid and in the course of the residual restrictions.
- If the problem is homogeneous (i.e., if the right-hand side of the linear system corresponding to the finest grid level vanishes), DiMEPACK uses dedicated smoothing functions in order to avoid unnecessary memory accesses.
- If the matrix to be factorized is symmetric and positive definite, we apply Cholesky's method instead of computing an LU decomposition. This approximately saves 50% of the corresponding floating-point operations.
- DiMEPACK tries to save multiply operations by factoring out identical stencil coefficients.

9.3.2.2 Locality Optimizations

Data layout optimizations. These techniques target the storage schemes which are used to arrange the data in memory, see Chapter 5. One of the most prominent layout optimization techniques is array padding, cf. Section 5.2. DiMEPACK uses appropriate paddings for all arrays involved in the computation.

[7] Marginal differences in the numerical results may occur due to a reordering of the arithmetic operations by an aggressively optimizing compiler and due to the fact that certain arithmetic rules such as the law of associativity do not hold for finite precision floating-point arithmetic.

Data access optimizations As was explained in Chapter 6, these techniques change the order in which the data is referenced. The objective is to enhance both spatial and temporal locality, yielding higher cache hit rates and, as a consequence, lower execution times. We have implemented the optimization techniques loop fusion as well as *one-dimensional* and *two-dimensional* loop blocking [Wei01]. The guiding principle of these data access optimizations is the use of a working set which is small enough to be kept in cache.

9.3.3 Performance of the DiMEPACK Library

The performance of the DiMEPACK library reflects the performance speedups for the individual computational kernels. See Section 7.2 and the references to the literature we have given therein. A comprehensive discussion of the performance of the DiMEPACK library exceeds the scope of this thesis. Instead, we refer to [Wei01].

9.4 Examples of Locality Metrics

9.4.1 Overview

As was explained in Section 2.5, profiling techniques and simulation approaches can be used to quantify the cache performance of application codes. Alternatively, *locality metrics* can be employed to predict cache efficiency as well as data access complexity. These machine-independent metrics are used in order to quantify the spatial or the temporal locality exhibited by the application code under consideration, while ignoring its actual computational complexity.

Some approaches to quantify the locality exhibited by an application use *stack histograms* in order to accurately predict its cache behavior in a machine-independent way. The idea is to gather information about the memory references of the code at compile time first and to guide locality-enhancing transformations afterwards. For this purpose, a trace of memory references is processed and the distances between the current reference and all previous references are determined and stored. For further details, see [CP03, Cas00] and the references therein. In particular, we point to [CPHL01] where an analytical cache modeling technique based on a new classification of cache misses is presented, and also to [GMM98] where the framework of *cache miss equations* is derived and employed. In Sections 9.4.2 and 9.4.3, we will present two additional locality metrics.

Note that some theoretically oriented research has focused on the development of so-called *cache-oblivious* algorithms, which do not depend on hardware parameters; e.g., cache size and cache line length. Asymptotically optimal cache-oblivious algorithms have been presented for matrix transpose, FFT, and sorting, for example. They are based on a set of simplifying assumptions concerning cache architectures [FLPR99, Kum03].

9.4.2 The Concept of Memtropy

The concept of *memtropy* to quantify the regularity of a sequence of memory references has been proposed in [Key00]. The memtropy μ of a sequence of memory addresses $m_0, \ldots, m_l$ is a real scalar value defined as follows:

$$\mu := \frac{1}{l} \sum_{i=1}^{l} \sum_{j=0}^{i-1} e^{-\alpha j^2} |m_i - m_{i-1-j}| \ . \tag{9.5}$$

The author further defines $m_0 := m_1$ in order to keep the formula simple[8].

[8] This means that the first memory reference incurs no penalty.

By definition, a set of cache-friendly ordered memory references should have lower memtropy than a poorly ordered one. In particular, the constant factor $\alpha > 0$ occurring in the exponent of the exponential term is machine-dependent and controls the impact of the memory distances between the currently referenced data item (located at address m_i) and all previous memory references m_j, $j < i$, on the memtropy of the sequence. The memtropy model respects both spatial and temporal locality. See [Key00] for details and for examples. In particular, an example of a grid-based update pattern based on loop blocking is provided.

9.4.3 Definition and Application of a Temporal Locality Metric

9.4.3.1 Motivation and Definition

An alternative locality metric has been proposed by Rüde [Rüd03]. Again, we assume a sequence $m_0, m_1, \ldots$ of memory addresses and a *cost function*

$$K : \mathbf{N} \to \mathbf{R}_0^+ \ . \tag{9.6}$$

Typically, the definition of K is physically motivated and K is chosen to model a "continuous memory hierarchy". The latter is supposed to approximate a real hardware design which is characterized by several discrete levels of memory. Therefore, reasonable choices for the cost function K cover

- $K(t) = t^{\frac{1}{3}}$, which is motivated by a model representing the data to be arranged in spheres that surround the computational core (i.e., the ALU) in 3D physical space, or
- $K(t) = \log t$, which represents a continuous "tree-like" memory architecture.

In order to keep the notation simple, we define for $i, j \in \mathbf{N}$

$$S_{i,j} := \{k \in \mathbf{N};\ i < k < j\} \tag{9.7}$$

to be the set of all natural numbers between i and j. The cost $C(m_j)$ for accessing the data item located at memory address m_j are then modeled as follows:

$$C(m_j) = \begin{cases} K\Big(|\{m_k;\ k \in S_{i,j},\ m_i = m_j,\ \forall \xi \in S_{i,j} :\ m_\xi \neq m_j\}|\Big) \ , & \text{if } \exists i : 0 \leq i < j,\ m_i = m_j \ , \\ \\ C_{\max}, & \text{otherwise} \ . \end{cases} \tag{9.8}$$

The locality metric defined in (9.8) can be interpreted as follows. If address m_j has been referenced previously (which corresponds to the first case), the minimum number of pairwise different intermediate memory accesses determines the cost $C(m_j)$ for accessing it anew. This approach represents the data item located at memory address m_j moving down the memory hierarchy; i.e., away from the ALU. How fast the data item moves away from the ALU is governed by the cost function K.

If, however, m_j has not been referenced before (which corresponds to the second case), $C(m_j)$ takes the fixed maximum value $C_{\max}$. This case models the data item being retrieved from the slowest level of the memory hierarchy; i.e., from main memory in our case.

Since the mutual distances between the memory addresses are not taken into consideration, this model can only be used to quantify temporal locality. The impact of the code exhibiting spatial locality (particularly through the concept of cache blocks) is completely ignored. This is a fundamental difference between the memtropy measure from Section 9.4.2 and the current approach.

9.4.3.2 Example: Optimization of Lexicographic SOR in 2D

In order to keep this example as simple as possible, we consider Laplace's equation $\Delta u = 0$ on the unit square and assume Dirichlet boundary conditions (see again Section 3.1). We further assume a discretization based on constant 5-point stencils on a regular 2D grid containing n nodes in each dimension. We apply SOR with a lexicographic ordering of the unknowns in order to solve the resulting linear system iteratively, see Section 3.2.4.

Note that we count read operations only. This is a reasonable approach since any approximation $u_i^{(k)}$ must be read first before the next iterate $u_i^{(k+1)}$ can be computed and written, cf. the SOR update scheme given by (3.15) in Section 3.2.4. In order to further simplify the model, we ignore the outermost layer of grid nodes representing the boundary. Since we assume the number n of grid points per dimension to be sufficiently large, this simplification is justifiable.

Furthermore, since we assume the number of SOR iterations to be large as well, we also ignore the cost of the first sweep over the grid (i.e., the cost of the first SOR iteration). This implies that the occurrence of compulsory cache misses does not have to be taken into account, cf. Section 2.4.4.

Additionally, we ignore the constant factor of 5 that relates the number of elementary update operations to the actual number of data accesses and, since we are finally interested in complexity orders, we neglect irrelevant constants.

Under these simplifying assumptions, the memory access cost of an elementary update operation can be expressed as follows. Each elementary update operation requires the five iterates corresponding to the current position of the 5-point stencil to be accessed, see Figure 9.3. The values for both the western neighbor node and the current node have been accessed in the previous update step; i.e., during the update of the western neighbor node. The values corresponding to the southern neighbor node and to the eastern neighbor node were accessed n update steps ago; i.e., when the eastern neighbor node of the southern neighbor node was updated for the last time. Finally, the approximation for the northern neighbor node was accessed n^2 update steps ago; i.e., when the northern neighbor node itself was updated for the last time during the previous SOR iteration.

As a consequence, the cost c_{elem} for an elementary update operation is given by

$$c_{\text{elem}} = K(n^2) + 2K(n) + 2K(1) \ . \tag{9.9}$$

If we assume $K(t) = t^{\frac{1}{3}}$, the memory access cost c_{sweep} for an SOR sweep over the entire grid

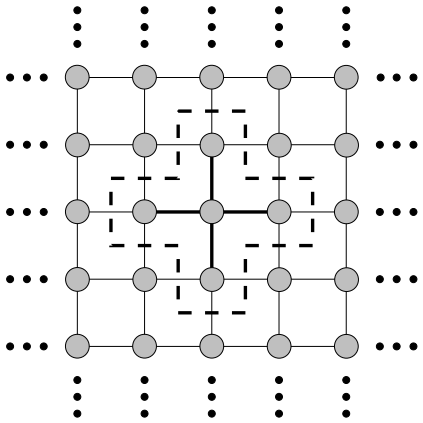


Figure 9.3: Elementary update operation based on a 5-point stencil.

comprising n^2 nodes therefore results as

$$c_{\text{sweep}} = n^2(K(n^2) + 2K(n) + 2K(1)) \in \mathcal{O}(n^{\frac{8}{3}}) \ . \tag{9.10}$$

It can be shown that, in the case of our model problem, the spectral radius $\rho(M)$ of the SOR iteration matrix M behaves like $1 - \mathcal{O}(n^{-1})$. See also Section 3.3.4. Hence, in order to reduce the algebraic error by a prescribed factor of ε, we need to perform $\nu = \nu(\varepsilon)$ SOR iterations, where ν is of order $\mathcal{O}(n)$. This follows from the requirement

$$\left(1 - \frac{c}{n}\right)^{\nu} < \varepsilon \ . \tag{9.11}$$

As a consequence, we obtain for the total memory access cost c_{total} of the SOR solver applied to the model problem that

$$c_{\text{total}} \in \mathcal{O}(n) \cdot \mathcal{O}(n^{\frac{8}{3}}) = \mathcal{O}(n^{\frac{11}{3}}) \ . \tag{9.12}$$

Now we consider the case of a line-wise blocked version of the SOR method, using the same ordering of the unknowns. This approach is similar to the *one-dimensional* blocking technique for red-black Gauss-Seidel presented in [Wei01] and to the 1-way blocking technique for Jacobi-type methods and CA we have described in Section 6.4.2.2.

If we block B successive SOR iterations into a single sweep over the grid ($B = B(n)$, $B << n$) and ignore the influence due to the special handling of the grid points located close to the boundary, the cost $c^{\text{bl}}_{\text{sweep}}$ of one such sweep will be

$$c^{\text{bl}}_{\text{sweep}} = n^2 \left(c_{\text{north}} + c_{\text{east}} + c_{\text{south}} + c_{\text{west}} + c_{\text{center}}\right) \ . \tag{9.13}$$

Figure 9.4 illustrates the case $B = 4$. This situation can be pictured as a block of four rows (indicated by the dashed horizontal lines) moving through the grid from bottom to top in a wavefront-like manner. For any grid point belonging to rows 1, 2, 3, and 4 of the current block, respectively, we must determine the numbers of grid nodes that have been updated since its corresponding neighboring nodes were accessed for the last time. Note that we only consider the numbers of nodes that have been updated at least once. It does not matter that some of them have been updated multiple times. Under the previous simplifying assumptions, these numbers of intermediate update operations can be expressed as shown in Table 9.1.

For general values of B, the cost for accessing the values at the grid nodes which correspond to

Block row	north	east	south	west	center
1	$(n-B)n$	Bn	$(B-1)n$	1	1
2	n	n	$(B-1)n$	1	1
3	n	n	$(B-1)n$	1	1
4	n	n	Bn	1	1

Table 9.1: Numbers of grid nodes which have been updated intermediately (possibly several times), line-wise blocked SOR, $B = 4$, cf. Figure 9.4.

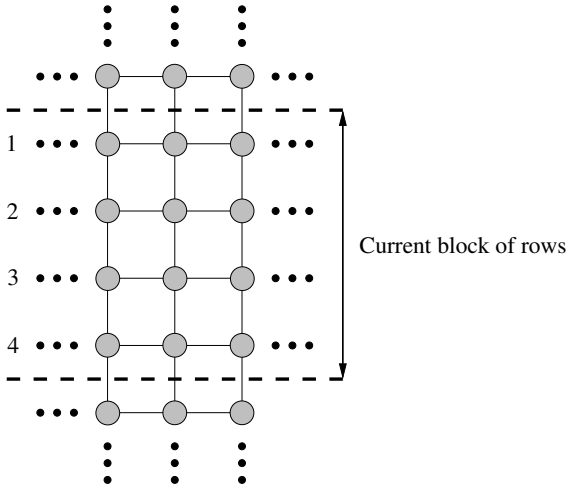


Figure 9.4: Line-wise blocked SOR algorithm, $B = 4$.

the current position of the 5-point stencil can be determined as follows:

$$c_{\text{north}} = K((n-B)n) + (B-1)K(n) \ , \tag{9.14}$$

$$c_{\text{east}} = K(Bn) + (B-1)K(n) \ , \tag{9.15}$$

$$c_{\text{south}} = (B-1)K((B-1)n) + K(Bn) \ , \tag{9.16}$$

$$c_{\text{west}} = BK(1) \ , \tag{9.17}$$

$$c_{\text{center}} = BK(1) \ . \tag{9.18}$$

Again, with $K(t) = t^{\frac{1}{3}}$ and $B << n$, we find that

$$c_{\text{sweep}}^{\text{bl}} \in \mathcal{O}(n^{\frac{8}{3}} + B^{\frac{4}{3}} n^{\frac{7}{3}}) \ . \tag{9.19}$$

Since, in the case of the line-wise blocked version of the SOR method, we need to perform only $\mathcal{O}(\frac{n}{B})$ sweeps over the grid in order to compute the same numerical approximation, we obtain for the total cost $c_{\text{total}}^{\text{bl}}$ of the blocked algorithm:

$$c_{\text{total}}^{\text{bl}} \in \mathcal{O}\left(\frac{n^{\frac{11}{3}}}{B} + B^{\frac{1}{3}} n^{\frac{10}{3}}\right) \ . \tag{9.20}$$

A comparison of (9.12) and (9.20) illustrates how the choice of $B = B(n)$ can influence the asymptotic memory access complexity of the algorithms. Note that, if B is chosen as a constant factor, the asymptotic memory access complexities of the blocked algorithm and the unblocked algorithm are the same.

If, however, we allow $B = B(n) \in \mathcal{O}(n^{\xi})$, the order of $c_{\text{total}}^{\text{bl}}$ becomes minimal, if both terms in (9.20) are of the same order; i.e., if

$$\frac{11}{3} - \xi = \frac{\xi}{3} + \frac{10}{3} \ , \tag{9.21}$$

which is equivalent to $\xi = \frac{1}{4}$. Hence, if we choose $B \in \mathcal{O}(n^{\frac{1}{4}})$, we find that

$$c_{\text{total}}^{\text{bl}} \in \mathcal{O}(n^{\frac{11}{3} - \frac{1}{4}}) = \mathcal{O}(n^{\frac{41}{12}}) \ . \tag{9.22}$$

If we compare this result with (9.12), we observe that, using this model, our blocking approaches can reduce the data access complexity by more than a constant factor.

A more detailed analysis — particularly of the influence of the choice K, the introduction of more complicated blocking techniques, and the extension to 3D — is beyond the scope of this thesis and left to future work.

9.5 Generation of Optimized Source Code Using ROSE

9.5.1 Overview of ROSE

The manual introduction of cache-based transformations is typically both tedious as well as error-prone. As was illustrated in the previous chapters, the application of code optimization techniques such as loop blocking can enormously increase the size of the code and often requires the treatment of special cases; e.g., close to the boundaries of the computational domain in the case of grid-based PDE solvers. Therefore, various research groups have concentrated on the automization of the introduction of code transformations. See [Qui00, KBC+01, MP99, GL99], for example.

This section briefly describes the structure and the functionality of the software package ROSE[9]. ROSE is a general framework to build source-to-source compilers (preprocessors) [Qui00, QSPK03, SQ03].

Such preprocessors can for example be defined to introduce a large variety of code transformations; e.g., cache-based optimizations such as loop blocking [QSMK04]. In [QSYdS03], the authors describe the translation of a serial C++ code including high-level abstractions into a parallel program using the ROSE framework. This translation consists of two phases. During the first phase, appropriate OpenMP directives are introduced. Afterwards, in the course of the second phase, these directives are translated into explicitly parallel C++ code, which is based on an OpenMP runtime library and can be compiled using a standard C++ compiler. According to [QSYdS03], this second phase is motivated by the current lack of OpenMP/C++ compiler infrastructure.

9.5.2 The Infrastructure of ROSE

Typically, large codes in scientific computing are based on the extensive use of software libraries. In the case of object-oriented frameworks, these libraries provide high-level abstractions; e.g., array classes, grid classes, or particle classes. Since the semantics of these abstractions are library-specific, they are unknown to the compiler during the compilation of the user's application. Therefore, it cannot optimize their use, which in turn often results in poor performance of the application. Note that this is a fundamental problem which is based on the fact that the abstractions provided by the software library are not part of the programming language definition itself.

ROSE represents a software framework to generate source-to-source compilers for the introduction of library-specific code transformations. Currently, ROSE supports the development of preprocessors which translate C++ source code into C++ source code again. ROSE is by no means restricted to generate preprocessors to perform optimizing transformations which improve data locality, for example. Alternatively, ROSE can also be employed to build source-to-source compilers which instrument code; e.g., by inserting calls to functions in profiling libraries or by integrating debugging support.

[9] Part of this work was carried out while the author was visiting the *Center for Applied Scientific Computing* at *Lawrence Livermore National Laboratory* as a temporary research employee in 2001 and 2002. This work was done under the supervision of Dr. D.J. Quinlan.

Obviously, the generation of library-specific optimizing preprocessors does not represent a temporary work-around which will finally be replaced by more sophisticated compiler technology. This research rather addresses an issue which necessarily results from the idea to enhance flexibility by building complex software from highly modular components; i.e., libraries. The approach behind the ROSE project can be understood as the treatment of an object-oriented library as a domain-specific language [QMPS02].

In the following, we will describe the current general structure of a source-to-source compiler which has been built using ROSE, see Figure 9.5. It is important to mention that any preprocessor built using ROSE still depends on the subsequent invocation of an optimizing compiler. Preprocessors based on ROSE are intended to perform high-level code transformations and do not address compiler back-end issues such as register allocation and machine code generation.

1. Building the EDG AST: The user code is parsed using the *EDG (Edison Design Group)* C++ compiler front-end[10]. This results in an internal representation as an abstract syntax tree (AST) [AK01].

2. Building the SAGE AST: Since the EDG interface is proprietary, the EDG AST is transformed into an AST which has a publically available interface. For this purpose, SAGE III, a modified and automatically generated version of the object-oriented SAGE++ framework [BBG+94], is used, see [SQ03]. SAGE III can be understood as an AST restructuring tool which is based on a collection of C++ classes to represent objects in C++ programs; e.g.,

   - all kinds of C++ statements, such as
     - class declaration and class definition statements,
     - function declaration and function definition statements,
     - `while` statements,
     - `if` statements,
     - etc.,
   - all kinds of C++ expressions, such as
     - unary `+` expressions,
     - binary `+` expressions,
     - binary `&&` expressions,
     - constant expressions,
     - variable expressions,
     - etc.,
   - etc.

3. Transforming the SAGE AST: The SAGE AST is transformed according to transformation rules which have to be specified by the library designer. Technically, these transformations are based on the application of a general tree traversal mechanism which is an essential part of the SAGE III framework. Typically, complex AST transformations are based on nested tree traversals which first query the original AST and then generate new AST fragments from source code strings by again invoking the compiler front-end. These new AST fragments are eventually inserted into the AST, possibly replacing its corresponding parts. The tree traversal mechanism used in ROSE is based on the use of inherited attributes and synthesized

[10] See `http://www.edg.com`.

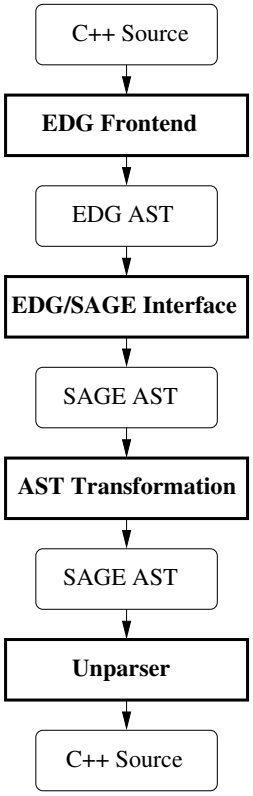


Figure 9.5: Structure of a C++ source-to-source compiler built using ROSE.

attributes. These data structures are employed in order to propagate information towards the leaves and towards the root of the AST, respectively [SQ03].

4. Unparsing the output: The AST is unparsed in order to generate the transformed C++ source code, which is written to a file.

5. Invoking the native C++ compiler: After the transformed AST has been unparsed into an output file, the native C++ compiler is invoked in order to generate object code or an executable file. Whether this last step is regarded as part of the preprocessor is a matter of interpretation; it is of course possible to integrate the invocation of the vendor-supplied C++ compiler into the source code of the preprocessor.

Chapter 10

Conclusions and Future Work

10.1 Conclusions

The continued improvements in processor performance are generally placing increasing pressure on the memory hierarchy. Therefore, we have developed and examined techniques in order to enhance the data locality exhibited by grid-based numerical computations. These techniques cover source-to-source transformations of the data layout and the data access order which may theoretically be automized. We have demonstrated that their application can lead to significant speedups of implementations of iterative numerical algorithms for linear systems and cellular automata (particularly the lattice Boltzmann method for simulating fluid flow problems) in both 2D and 3D. For the case of iterative linear solvers, we have primarily focused on the asymptotically optimal class of multigrid methods.

In the course of the optimization process, the selection of appropriate tuning parameters and suitable compiler options represents a crucial and nontrivial task which is highly dependent on the underlying platform. Therefore, our tuning efforts are based on the application of empirical software optimization techniques such as exhaustively searching the parameter space. This has proven to be a promising way to achieve high code efficiency for a wide range of numerical software libraries. Such heuristic approaches can be seen as software engineering methodologies to keep pace with the enormous evolution speed of computer architectures and to generate portable, yet highly optimized code.

As an example of a more fundamental algorithmic modification, we have further developed an approach towards novel multigrid methods which are inherently cache-aware through the use of patch processing techniques. Our patch-adaptive multigrid algorithm is derived from the theory of the fully adaptive multigrid method. If the patch sizes are chosen appropriately, our numerical scheme leads to a high utilization of the cache hierarchy. Additionally, it can be demonstrated that it behaves more robustly in the presence of operator-based singularities or perturbations from boundary conditions. A detailed analysis of our approach is part of the ongoing research in the DiME project.

It can be observed that, particularly for the case of iterative linear solvers, even the performance of the cache-tuned implementations is still far below the theoretically available peak performance of the respective target platform. Further efficiency speedups may be achieved through the application of more sophisticated optimization techniques concerning hardware improvements, progress in compiler technology, and the design of novel cache-aware algorithms. Additionally, the increas-

ing demand for performance may motivate the development of architecture-oriented programming languages for high performance computing.

However, a fundamental performance problem with any implementation of the grid-based iterative linear solvers we have considered results from their relatively low computational intensities; i.e., these methods are characterized by extremely low ratios of floating-point operations to data accesses. As a consequence, their implementations will continue to represent a challenge for the performance of hierarchical memory architectures. This effect is less dramatic for implementations of the lattice Boltzmann method due to the involved higher computational intensities.

We believe that the interaction of the CPU core and the memory hierarchy raises the need for integrated algorithmic complexity models which consider both operations to be executed as well as data accesses and their locality. We have included a brief presentation of memory-centric complexity models which aim at quantifying the locality of data accesses while ignoring arithmetic operations.

10.2 Practical Applications of our Work

10.2.1 Bioelectric Field Simulations

Our optimization techniques have successfully been integrated into a code which implements an SOR algorithm in order to solve potential equations occurring in the area of biomedical engineering. The overall goal of our target application is the localization of functionally important brain areas from electroencephalographic measurements of the potential field on the human scalp. These areas are typically modeled using dipoles. The most time-consuming part of the target application is the successive iterative solution of numerous so-called *forward problems*: based on assumptions concerning the position and the orientation of the dipole, the resulting electric potentials at the electrodes, which are located on the surface of the patient's head, need to be computed.

Solving such a forward problem means solving a PDE of the type $\nabla \cdot (\sigma \nabla u) = f$, where the computational domain is the patient's head. The tensor $\sigma = \sigma(x, y, z)$ specifies the conductivity of the different brain areas and the right-hand side $f = f(x, y, z)$ characterizes the electrical sources and sinks inside the head; i.e., the dipole. The solution $u = u(x, y, z)$ represents the electric potential field. We refer to [MV03] for further details.

The integration of our optimization techniques into the SOR code for the solution of these forward problems has led to a speedup of more than a factor of 3 on a variety of platforms. A detailed discussion of this code tuning process is beyond the scope of this thesis. Instead, we refer to [Zet00].

10.2.2 Chip Layout Optimization

Our DiMEPACK library has been integrated into a software framework for chip layout optimization [OJ04]. The objective of this framework is to automatically optimize the placement of cells in VLSI circuits. This optimization is based on requirements concerning timing behavior and heat distribution, amongst others.

An iterative numerical approach towards this problem is explained in detail in [EJ98]. In each iteration of the algorithm, Poisson's equation needs to be solved on a regular 2D grid, subject to appropriate boundary conditions which can be approximated by Neumann boundaries and thus handled by DiMEPACK, see Section 9.3.1.

The solution of this problem represents the current potential which in turn defines the driving forces acting on the cells and causing them to spread evenly across the layout area. The iteration process terminates as soon as the current cell distribution meets some given termination criterion.

10.3 Suggestions for Future Work

10.3.1 Integration of Cache Optimizations into Parallel Applications

So far, we have developed and investigated techniques in order to optimize the data locality exhibited by numerical codes on individual computing nodes based on hierarchical memory architectures. Future research must necessarily address the integration of these techniques into parallel applications; e.g., running on clusters of workstations. As was mentioned in the beginning in Section 1.2.1, high performance parallel computing comprises both the tuning of parallel efficiency — in terms of reasonable scalability of the numerical application — as well as (cache-)optimized code execution on the individual nodes.

In particular, the HHG framework which has been described in Section 9.1.3.2 represents an ideal parallel software infrastructure to be aimed at since the underlying meshes are composed of highly regular building blocks. For this purpose, our tuning techniques for iterative linear solvers must be extended to further types of discretization stencils; e.g., resulting from triangular elements in 2D or from tetrahedra and prisms in 3D.

There is no doubt about the fact that sophisticated software engineering approaches — such as the automatic introduction of cache-based optimizations (cf. Section 9.5) or the use of domain-specific languages to generate efficient code — will continue to play a crucial role in the development and the maintenance of portable and flexible high performance applications.

10.3.2 On-the-fly Assembly of Linear Equations

Due to the increasing gap between CPU speed and memory performance, moving data has become more costly than processing data. This observation raises the question of whether efficient trade-offs between repeatedly computing and repeatedly accessing data must be striven for. This is particularly the case for the discretization of "sufficiently simple" differential operators.

As an example, we consider the scalar elliptic differential operator $L := -\nabla(a\nabla u)$, $a = a(x, y, z) > 0$, in 3D and assume a finite element discretization involving trilinear cubic elements [Bra97]. For grid nodes in the interior of the computational domain, this approach leads to (variable) 27-point stencils. As a consequence, the update of the numerical approximation corresponding to an interior grid node using a Gauss-Seidel iteration scheme (cf. (3.13) in Section 3.2.3) requires 27 stencil weights (i.e., matrix entries) to be accessed, in addition to the approximations at the respective 26 neighboring grid nodes and the corresponding value of the right-hand side.

In many cases, the values of the function a are obtained from physical measurements or sampled at quadrature points. They are therefore often assumed to be constant within each of the cubic elements. Hence, the calculation of the 27 stencil weights for an interior grid point is based on the values of a within its eight neighboring elements. Hereby, the stencil weights are determined by the respective entries of the corresponding element stiffness matrices, see [JL01], for example. If the latter can be computed efficiently (involving appropriate numerical quadrature rules), this could potentially lead to enhanced execution speed due to reduced memory traffic.

The examination of such strategies which trade increased computational work for reduced memory costs is left to future research. Algorithmic transformations of this kind can be interpreted as high-level cache optimizations which are a long way from being automatically introduced by an optimizing compiler.

10.3.3 Extension of the Patch-Adaptive Multigrid Scheme

The patch-adaptive multigrid algorithm we have developed in our DiME project and presented in Chapter 8 is based on the theory of the fully adaptive multigrid method. As was mentioned in Section 8.4.1, our scheme still maintains a conservative cycling strategy based on a level-oriented processing of the grid hierarchy.

However, this is by no means required by the theory. Instead, more involved adaptive cycling strategies can be implemented. Such strategies involve inter-grid transfer operators which are restricted to certain parts of the grids only. This enhanced flexibility may be exploited to further improve the numerical efficiency of the patch-adaptive multigrid method since unnecessary residual calculations can be omitted in those parts of the grids where the approximations have not changed since the last time the residual was determined.

In addition, as was indicated in Section 8.1, future research could address the integration of adaptive mesh refinement and adaptive relaxation in order to radically exploit the benefits guaranteed by the theory of the fully adaptive multigrid method.

Part V

Appendix

Appendix A

Machine Specifications

Platform A

- Workstation name: DEC PWS 500au
- CPU: Alpha 21164 (EV 5), 500 MHz
- L1: 8 kB data, 8 kB instructions (both direct mapped, on-chip)
- L2: 96 kB (3-way set-associative, on-chip)
- L3: 4 MB (direct-mapped, off-chip)
- TLB: data: 64 entries (w/ 8 kB pages), instructions: 48 entries (w/ 8 kB pages)
- Cf. [Dig97]

Platform B

- Workstation name: Compaq XP 1000 Professional
- CPU: Alpha 21264 (EV 6), 500 MHz
- L1: 64 kB data, 64 kB instructions (both 2-way set-associative, on-chip)
- L2: 4 MB (direct-mapped, off-chip)
- TLB: data: 128 entries (w/ 8 kB pages), instructions: 128 entries (w/ 8 kB pages)
- Cf. [Com99]

Platform C

- CPU: AMD Athlon K75, 700MHz
- L1: 64 kB data, 64 kB instructions (both 2-way set-associative, on-chip)
- L2: 512 kB (2-way set-associative, off-chip)
- TLB: L1 data: 24 entries (w/ 4 kB pages), L1 instructions: 16 entries (w/ 4 kB pages), L2 data: 256 entries (w/ 4 kB pages), L2 instructions: 256 entries (w/ 4 kB pages)

Platform D

- CPU: AMD Athlon XP 2400+, 2.0 GHz
- L1: 64 kB data, 64 kB instructions (both 2-way set-associative, on-chip)
- L2: 256 kB (16-way set-associative, on-chip)

- TLB: 32 entries (L1), 256 entries (L2)
- TLB: L1 data: 32 entries (w/ 4 kB pages), L1 instructions: 16 entries (w/ 4 kB pages), L2: 256 entries (w/ 4 kB pages)
- Cf. [AMD03]

Platform E

- CPU: AMD Opteron, 1.6 GHz
- L1: 64 kB data, 64 kB instructions (both 2-way set-associative, on-chip)
- L2: 1 MB (16-way set-associative, on-chip)
- TLB: L1 data: 32 entries (w/ 4 kB pages), L1 instructions: 32 entries (w/ 4 kB pages), L2: 512 entries (w/ 4 kB pages)
- Cf. [AMD03]

Platform F

- CPU: Intel Pentium 4, 2.4 GHz
- L1: 8 kB data (4-way set-associative, on-chip),
  16 kB instructions (8-way set-associative, on-chip)
- L2: 512 kB (8-way set-associative, on-chip)
- TLB: data: 64 entries (w/ 4 kB pages), instructions: 64 entries (w/ 4 kB pages)
- Cf. [Int02]

Platform G

- Workstation: HP zx6000
- CPU: Intel Itanium 2, 900 Mhz
- L1: 16 kB data, 16 kB instructions (both 4-way set-associative, on-chip)
- L2: 256 kB (8-way set-associative, on-chip)
- L3: 3 MB (12-way set-associative, on-chip)
- TLB: L1 data: 32 entries (w/ 4 kB pages), L1 instructions: 32 entries (w/ 4 kB pages),
  L2 data: 128 entries (w/ 4 kB pages), L2 instructions: 128 entries (w/ 4 kB pages)
- Cf. [Int03]

Appendix B

German Parts

B.1 Inhaltsverzeichnis

1 Einleitung **1**

1.1 Umfang der Arbeit 1
1.2 Übersicht — Motivation und Beiträge 1
1.2.1 Effizienzoptimierung numerischer Anwendungen 1
1.2.2 Zielalgorithmen 2
1.2.3 Optimierungsansätze 3
1.2.4 Das DiME-Projekt 3
1.3 Gliederung 4

I Speicherhierarchien und numerische Algorithmen **5**

2 Speicherhierarchien **7**

2.1 Übersicht: Steigerung der CPU-Leistung 7
2.2 Aufbau von Speicherhierarchien 7
2.2.1 Motivation 7
2.2.2 CPU-Register, Cache-Speicher und Hauptspeicher 8
2.2.3 Translation Lookaside Buffers 10
2.3 Das Lokalitätsprinzip 10
2.4 Eigenschaften von Cache-Architekturen 11
2.4.1 Cache-Zeilen und Satz-Assoziativität 11
2.4.2 Ersetzungsstrategien 11
2.4.3 Schreibstrategien 12
2.4.4 Klassifikation von Cache-Fehlzugriffen 13
2.5 Analyse der Cache-Effizienz 13
2.5.1 Code Profiling — Techniken und Werkzeuge 13
2.5.1.1 Hardware Performance Counters 13
2.5.1.2 Instrumentierung 14
2.5.2 Simulationsbasierte Analyse der Cache-Effizienz 15

3 Numerische Algorithmen I — Iterative lineare Löser 17
3.1 Vorbereitung: Diskretisierung partieller Differentialgleichungen . . . 17
3.2 Elementare lineare Löser . . . 18
3.2.1 Übersicht: direkte Verfahren vs. iterative Verfahren . . . 18
3.2.1.1 Vorbereitungen . . . 18
3.2.1.2 Direkte Verfahren . . . 19
3.2.1.3 Iterative Verfahren . . . 19
3.2.1.4 Warum wir uns auf iterative Verfahren konzentrieren . . . 22
3.2.2 Die Methode von Jacobi . . . 23
3.2.3 Die Methode von Gauß-Seidel . . . 23
3.2.4 Jacobi-Überrelaxation und sukzessive Überrelaxation . . . 25
3.2.5 Bemerkungen . . . 26
3.3 Mehrgitterverfahren . . . 27
3.3.1 Übersicht . . . 27
3.3.2 Vorbereitungen . . . 28
3.3.3 Die Residuengleichung . . . 29
3.3.4 Konvergenzverhalten elementarer iterativer Methoden . . . 29
3.3.5 Aspekte von Mehrgitterverfahren . . . 30
3.3.5.1 Grobgitterrepräsentation der Residuengleichung . . . 30
3.3.5.2 Gittertransferoperatoren . . . 31
3.3.5.3 Grobgitteroperatoren . . . 33
3.3.5.4 Zweigitterverfahren . . . 33
3.3.5.5 Verallgemeinerung auf Mehrgitterverfahren . . . 34
3.3.5.6 Bemerkungen zur Konvergenzanalyse von Mehrgitterverfahren . . . 37
3.3.5.7 Full Approximation Scheme . . . 37
3.3.5.8 Geschachtelte Iteration und volles Mehrgitter . . . 40

4 Numerische Algorithmen II — zelluläre Automaten 43
4.1 Formale Spezifikation zellulärer Automaten . . . 43
4.2 Die Lattice-Boltzmann-Methode . . . 45
4.2.1 Einleitung . . . 45
4.2.2 Diskretisierung der Boltzmann-Gleichung . . . 46
4.2.3 Die LBM in 2D . . . 47
4.2.4 Die LBM in 3D . . . 49
4.2.5 Randbedingungen . . . 49
4.2.6 Abstrakte Beschreibung der LBM . . . 51
4.2.7 LBM-Modellproblem: Lid-Driven Cavity . . . 52

II Techniken der Code-Optimierung für Speicherhierarchien 55

5 Optimierungen der Datenanordnung 57
5.1 Grundlagen . . . 57
5.2 Array Padding . . . 57
5.2.1 Motivation und Prinzip . . . 57
5.2.2 Intra-array Padding in 2D . . . 59
5.2.3 Intra-array Padding in 3D . . . 59

5.2.3.1 Der Standardansatz . . . 59
5.2.3.2 Der Nichtstandardansatz . . . 60
5.2.3.3 Beispiel: nichtstandardmäßiges intra-array Padding für Gauß-Seidel 62
5.3 Cache-optimierte Datenanordnungen . . . 64
5.3.1 Array Merging . . . 64
5.3.2 Cache-optimierte Datenstrukturen für iterative lineare Löser . . . 64
5.3.3 Cache-optimierte Datenstrukturen für zelluläre Automaten . . . 66
5.4 Gitterkompression für zelluläre Automaten und Jacobi-artige Verfahren . . . 67

6 Datenzugriffsoptimierungen 71
6.1 Grundlagen . . . 71
6.2 Schleifenvertauschung . . . 72
6.2.1 Motivation und Prinzip . . . 72
6.2.2 Schleifenvertauschung bei iterativen linearen Lösern . . . 73
6.2.3 Schleifenvertauschung bei zellulären Automaten und Jacobi-artigen Verfahren 73
6.3 Schleifenverschmelzung . . . 74
6.3.1 Motivation und Prinzip . . . 74
6.3.2 Anwendung von Schleifenverschmelzung auf rot-schwarz Gauß-Seidel . . . 74
6.3.3 Anwendung von Schleifenverschmelzung auf die LBM . . . 77
6.4 Blocken der Schleifen . . . 77
6.4.1 Motivation und Prinzip . . . 77
6.4.2 Blocken der Schleifen in 2D . . . 79
6.4.2.1 Geblockte Varianten von rot-schwarz Gauß-Seidel . . . 79
6.4.2.2 Geblockte Varianten zellulärer Automaten und Jacobi-artiger Verfahren . . . 79
6.4.3 Blocken der Schleifen in 3D . . . 83
6.4.3.1 Geblockte Varianten von rot-schwarz Gauß-Seidel . . . 83
6.4.3.2 Geblockte Varianten zellulärer Automaten und Jacobi-artiger Verfahren . . . 92
6.5 Weitere Datenzugriffsoptimierungen . . . 97
6.5.1 Prefetching von Daten . . . 97
6.5.2 Kopieren von Daten . . . 98

7 Experimentelle Ergebnisse 101
7.1 Einführende Bemerkungen . . . 101
7.1.1 Die Kombination von Effizienz und Flexibilität . . . 101
7.1.2 Zweck dieses Kapitels . . . 102
7.2 Iterative Verfahren auf 2D Gittern . . . 102
7.3 Iterative Verfahren auf 3D Gittern . . . 102
7.3.1 Beschreibung der Implementierungen . . . 102
7.3.2 Effizienzanalysen . . . 104
7.3.2.1 Analyse der Standardmehrgitterimplementierung . . . 104
7.3.2.2 Vergleich verschiedener Datenanordnungen . . . 107
7.3.2.3 Vergleich verschiedener Schleifenreihenfolgen . . . 109
7.3.2.4 Effizienz der Mehrgitterimplementierungen — allgemeiner Fall . . . 111

7.3.2.5 Effizienz der Mehrgitterimplementierungen — Spezialfall: konstante Koeffizienten . . . 120
7.4 Die LBM in 2D . . . 121
7.4.1 Beschreibung der Implementierungen . . . 121
7.4.2 Effizienzanalysen . . . 123
7.5 Die LBM in 3D . . . 128
7.5.1 Beschreibung der Implementierungen . . . 128
7.5.2 Effizienzanalysen . . . 128
7.6 Zusammenfassung dieses Kapitels . . . 132

III Entwurf inhärent Cache-effizienter Mehrgitteralgorithmen 135

8 Patch-Adaptive Mehrgitterverfahren 137
8.1 Übersicht . . . 137
8.2 Adaptive Relaxation . . . 138
8.2.1 Übersicht . . . 138
8.2.2 Definitionen und elementare Ergebnisse . . . 138
8.2.3 Algorithmische Beschreibung der adaptiven Relaxation . . . 140
8.3 Adaptive Relaxation in Mehrgitterverfahren . . . 141
8.3.1 Übersicht . . . 141
8.3.2 Vorbereitungen . . . 142
8.3.3 Das erweiterte lineare System . . . 143
8.3.3.1 Nichteindeutige Darstellung von Vektoren auf feinen Gittern . . . 143
8.3.3.2 Herleitung des erweiterten linearen Systems . . . 144
8.3.3.3 Eigenschaften der Matrix des erweiterten linearen Systems . . . 145
8.3.4 Iterative Lösung des erweiterten linearen Systems . . . 146
8.4 Einführung der Patch-Adaptivität . . . 146
8.4.1 Übersicht über das Patch-adaptive Mehrgitterverfahren . . . 146
8.4.2 Patch-Design . . . 148
8.4.3 Unmittelbare Konsequenzen des Patch-adaptiven Ansatzes . . . 150
8.4.3.1 Reduktion von Granularität und Overhead . . . 150
8.4.3.2 Erhöhte Relaxationsrate . . . 150
8.4.4 Patch-Verarbeitung . . . 151
8.4.4.1 Definition des skalierten Patch-Residuums . . . 151
8.4.4.2 Patch-Relaxation . . . 151
8.4.5 Aktivierung von Nachbar-Patches . . . 152
8.4.5.1 Problembeschreibung . . . 152
8.4.5.2 Änderungen skalierter Residuen . . . 152
8.4.5.3 Relaxations-/Aktivierungsstrategien . . . 152
8.4.6 Algorithmische Beschreibung der Patch-adaptiven Relaxation . . . 154
8.4.7 Bemerkungen zu Implementierungsdetails . . . 156
8.5 Experimentelle Ergebnisse . . . 157
8.5.1 Beschreibung des Modellproblems . . . 158
8.5.1.1 Definition . . . 158
8.5.1.2 Analytische Lösung . . . 158

8.5.1.3 Diskretisierung mittels finiter Elemente . . . 159
8.5.2 Numerische Tests . . . 161
8.5.2.1 Vorbereitungen . . . 161
8.5.2.2 Ergebnisse . . . 164

IV Verwandte Arbeiten, Zusammenfassung und Ausblick 167

9 Verwandte Arbeiten — ausgewählte Kapitel 169
9.1 Optimierungstechniken für numerische Software . . . 169
9.1.1 Überblick . . . 169
9.1.2 Automatische Effizienzoptimierung numerischer Software . . . 169
9.1.3 Weitere Optimierungstechniken für gitterbasierte Algorithmen . . . 170
9.1.3.1 Cache-Optimierungen für Mehrgitterverfahren auf unstrukturierten Gittern . . . 170
9.1.3.2 Hierarchisch-hybride Gitter . . . 171
9.1.3.3 Cache-effiziente parallele Gebietszerlegungsansätze . . . 172
9.1.3.4 Cache-effiziente parallele Implementierung zellulärer Automaten . . . 173
9.2 Cache-Optimierungen für die BLAS und LAPACK . . . 173
9.2.1 Überblick über die BLAS und LAPACK . . . 173
9.2.2 Steigerung der Cache-Effizienz der BLAS . . . 173
9.2.3 Blockalgorithmen in LAPACK . . . 175
9.3 Die Bibliothek DiMEPACK . . . 176
9.3.1 Funktionalität der Bibliothek DiMEPACK . . . 176
9.3.2 Optimierungstechniken . . . 177
9.3.2.1 Arithmetische Optimierungen . . . 177
9.3.2.2 Lokalitätsoptimierungen . . . 177
9.3.3 Effizienz der Bibliothek DiMEPACK . . . 178
9.4 Beispiele für Lokalitätsmetriken . . . 178
9.4.1 Übersicht . . . 178
9.4.2 Das Memtropy-Konzept . . . 178
9.4.3 Definition und Anwendung einer Metrik für temporale Lokalität . . . 179
9.4.3.1 Motivation und Definition . . . 179
9.4.3.2 Beispiel: Optimierung von lexikographischem SOR in 2D . . . 180
9.5 Generierung optimierten Quell-Codes mittels ROSE . . . 183
9.5.1 Überblick über ROSE . . . 183
9.5.2 Die Infrastruktur von ROSE . . . 183

10 Abschluss und zukünftige Arbeiten 187
10.1 Zusammenfassung . . . 187
10.2 Praktische Anwendungen unserer Arbeit . . . 188
10.2.1 Simulation bioelektrischer Felder . . . 188
10.2.2 Chip-Layout-Optimierung . . . 188
10.3 Vorschläge für zukünftige Arbeiten . . . 189
10.3.1 Integration von Cache-Optimierungen in parallele Anwendungen . . . 189
10.3.2 Dynamisches Aufstellen linearer Gleichungen . . . 189
10.3.3 Erweiterung der Patch-adaptiven Mehrgittermethode . . . 190

V Anhang **191**

A Spezifikation der Architekturen **193**

B Abschnitte in deutscher Sprache **195**

B.1 Inhaltsverzeichnis 195

B.2 Zusammenfassung 200

B.3 Einleitung 201

C Lebenslauf **205**

B.2 Zusammenfassung

Um die Auswirkungen der sich stets vergrößernden Kluft zwischen Prozessorgeschwindigkeit und Hauptspeicherleistung auf die Ausführungszeiten von Anwenderprogrammen zu mindern, werden in heutigen Rechnerarchitekturen meist hierarchische Speicherstrukturen verwendet, die mehrere Schichten von Cache-Speichern umfassen. Programme können nur dann effizient ausgeführt werden, wenn die zu Grunde liegende Speicherarchitektur berücksichtigt wird. Dies trifft insbesondere auf rechenintensive Software zu.

Unglücklicherweise sind gegenwärtige Compiler nicht in der Lage, komplizierte Cache-orientierte Programmtransformationen durchzuführen. Infolgedessen obliegt ein Großteil der aufwändigen und fehleranfälligen Optimierungsarbeit dem Software-Entwickler. Im Falle einer parallelen numerischen Applikation, die beispielsweise auf einem Workstation-Cluster läuft, stellt die Optimierung hinsichtlich der hierarchischen Speicherstruktur einen wichtigen Aspekt dar, der zusätzlich zu den typischen Zielen der Parallelisierung wie der effizienten Lastverteilung und der Minimierung des Kommunikationsaufwands berücksichtigt werden muss.

Die Optimierung der Speicherhierarchienutzung umfasst ein weites Spektrum, welches sich von Hardware-basierten Technologien (wie beispielsweise Daten-Prefetching-Mechanismen) über Compiler-basierte Programmtransformationen (wie etwa der Restrukturierung von Schleifen) bis hin zu grundlegenden algorithmischen Modifikationen erstreckt. Letztere können beispielsweise auf einer Vergrößerung des Rechenaufwands und einer damit verbundenen Reduktion der Speicherzugriffskosten beruhen und sind weit von einer automatischen Integration durch einen Compiler entfernt. Das Erreichen sowohl von hoher Effizienz zur Laufzeit als auch von Portabilität stellt dabei eine Aufgabe an das Software Engineering dar, die grundsätzlich beim Entwurf numerischer Bibliotheken bewältigt werden muss.

Im Rahmen dieser Dissertation erläutern wir Ansätze zur Optimierung der Datenlokalität von Implementierungen gitterbasierter numerischer Algorithmen. Insbesondere befassen wir uns mit Mehrgitterverfahren auf strukturierten Gittern sowie mit zellulären Automaten, jeweils sowohl in 2D als auch in 3D. Als repräsentatives Beispiel zellulärer Automaten betrachten wir die Lattice-Boltzmann-Methode zur Simulation strömungsmechanischer Probleme.

Desweiteren stellen wir einen neuartigen Ansatz im Hinblick auf inhärent Cache-effiziente Mehrgittermethoden vor. Unser Algorithmus, der auf der Theorie der volladaptiven Mehrgittermethode basiert, verwendet eine Patch-adaptive Verarbeitungsstrategie. Er ist folglich durch eine hohe Nutzung der Cache-Hierarchie gekennzeichnet. Darüber hinaus arbeitet unser Verfahren infolge der adaptiven Update-Strategie robuster als klassische Mehrgitteralgorithmen im Falle singulär gestörter Probleme.

B.3 Einleitung

Umfang der Arbeit

Diese Arbeit stellt einen Beitrag zum Forschungsgebiet des wissenschaftlichen Hochleistungsrechnens dar. Zusammenfassend gesagt behandelt die vorliegende Dissertation architekturgetriebene Effizienzoptimierungen rechenintensiver Programme. Insbesondere stehen die folgenden Aspekte im Vordergrund:

- Speicherhierarchieoptimierungen für rechenintensive Programme
- Architekturorientierte Entwicklung effizienter numerischer Verfahren
- Effizienzanalyse und Profiling der Speicherhierarchie
- Entwicklung flexibler und effizienter numerischer Software

Übersicht — Motivation und Beiträge

Effizienzoptimierung numerischer Anwendungen

Zum Zwecke der Effizienzsteigerung einer parallelen numerischen Applikation ist es unumgänglich, zwei miteinander in Beziehung stehende Optimierungsziele anzustreben, die jeweils fundierte Kenntnisse sowohl des Algorithmus als auch der Zielarchitektur erfordern. Zum einen ist es notwendig, den Parallelisierungsaufwand selbst zu minimieren. Dies beinhaltet typischerweise die Wahl geeigneter paralleler numerischer Algorithmen und Lastverteilungsstrategien sowie die Minimierung des Kommunikationsaufwands mittels Reduktion der Kommunikation und Verbergen von Netzwerklatenz und -bandbreite. Erheblicher Forschungsaufwand hinsichtlich der Optimierung paralleler Effizienz wurde bislang und wird auch weiterhin betrieben.

Zum anderen ist es entscheidend, die einzelnen parallelen Ressourcen so effizient wie möglich zu nutzen, etwa indem die maximal mögliche Rechenleistung auf jedem Knoten einer parallelen Umgebung erzielt wird. Dies trifft insbesondere auf Systeme mit verteiltem Speicher zu, wie sie in Clustern handelsüblicher Workstations verwendet werden, die mittels schneller Verbindungsnetzwerke miteinander kommunizieren. Üblicherweise basieren diese Maschinen auf hierarchischen Speicherarchitekturen, die verwendet werden, um die Auswirkungen der sich stets vergrößernden Kluft zwischen Prozessorgeschwindigkeit und Hauptspeicherleistung zu mildern.

Gemäß dem Moore'schen Gesetz aus dem Jahr 1975 verdoppelt sich die Anzahl der Transistoren auf einem Silizium-Chip innerhalb von 12 bis 24 Monaten. Diese Vorhersage traf seither während eines Zeitraums von fast drei Jahrzehnten beachtlich genau zu. Sie führte zu einer Steigerung der CPU-Leistung von etwa 55% pro Jahr. Im Gegensatz dazu hat sich die DRAM-Technologie eher langsam entwickelt. Hauptspeicherlatenz und Speicherbandbreite verbesserten sich um etwa 5% bzw. 10% pro Jahr [HP03]. Die *International Technology Roadmap for Semiconductors*[1] prognostiziert, dass dieser Trend weiterhin anhalten und die Kluft zwischen Prozessorgeschwindigkeit und Hauptspeicherleistung weiter anwachsen wird, bevor technologische Grenzen erreicht werden. Eine reichhaltige Sammlung von Beiträgen, die zuküftige Trends im Bereich Mikro- und Nanoelektronik sowie deren Beziehung zum Moore'schen Gesetz vorhersagen und erläutern, findet sich in [CiS03].

Die in der vorliegenden Dissertation beschriebene Forschungsarbeit befasst sich mit dem zuvor genannten zweiten Optimierungsziel, der Optimierung der Effizienz der einzelnen Rechenknoten.

[1] Siehe `http://public.itrs.net`.

Insbesondere erläutern und analysieren wir Techniken, die auf die Optimierung der Daten-Cache-Effizienz auf Architekturen mit hierarchischem Speicher abzielen. Die zu Grunde liegende Eigenschaft unserer Ansätze besteht also darin, Datenlokalität auszunutzen und zu optimieren. Demzufolge bedingt deren Anwendung sowohl eine Reduktion von Hauptspeicherzugriffen als auch eine Verminderung der Cache-Fehlzugriffe. Dies impliziert eine unmittelbare Verkürzung der Ausführungszeiten numerischer Applikationen, deren Effizienz hauptsächlich durch Speicherlatenz und Speicherbandbreite beschränkt wird.

Unglücklicherweise sind aktuelle optimierende Compiler nicht in der Lage, Folgen komplizierter und problemabhängiger Cache-basierter Programmtransformationen durchzuführen. Infolgedessen liefern sie selten die Leistung, die der Anwender erwartet. Ein Großteil der aufwändigen und fehleranfälligen Arbeit der Optimierung der Knoteneffizienz obliegt daher dem Software-Entwickler. Nur die richtige Kombination aus hoher paralleler Effizienz und optimierter Knotenleistung führt zu minimalen Programmausführungszeiten. Entwickler und Anwender paralleler Applikationen scheinen diesen Aspekt zeitweise zu vernachlässigen.

An dieser Stelle ist erwähnenswert, dass typische Schleifenkörper, die in numerischen Anwendungen auftreten, relativ klein sind und demnach selten die Kapazität heute üblicher Instruktions-Caches überschreiten. Demzufolge stellt die Optimierung der Instruktions-Cache-Effizienz kein Hauptproblem im Bereich des wissenschaftlichen Hochleistungsrechnens dar. Während weitere Arbeiten sich zwar mit der Optimierung der Zugriffe auf Externspeicher beschäftigen (wie z.B. Festplattenlaufwerke), stellt auch dies kein vordergründiges praktisches Problem dar [KW03].

Zielalgorithmen

Im Rahmen dieser Arbeit betrachten wir zwei algorithmisch ähnliche Klassen numerischer Verfahren, iterative lineare Löser und zelluläre Automaten. Diese Algorithmen zeichnen sich durch viele wiederholte Durchläufe der potentiell sehr großen zu Grunde liegenden Datensätze aus.

Als erstes präsentieren und erläutern wir Techniken zur Lokalitätsoptimierung iterativer Lösungsverfahren für große dünn besetzte lineare Gleichungssysteme. Häufig treten derartige Probleme im Rahmen der numerischen Behandlung partieller Differentialgleichungen auf. Wir betrachten zunächst elementare iterative Verfahren. Jedoch ist es keinesfalls ausreichend, sich der Optimierung von Algorithmen zu widmen, die aus mathematischer Sicht vergleichsweise ineffizient sind. Minimale Ausführungszeiten können verständlicherweise nur unter Verwendung optimierter Implementierungen numerisch optimaler Algorithmen erwartet werden.

Daher befassen wir uns im Anschluss mit der komplizierteren Klasse der Mehrgitterverfahren. Diese Algorithmen basieren typischerweise auf elementaren Iterationsverfahren. Man kann zeigen, dass sie sich asymptotisch optimal verhalten. Dies bedeutet, dass die Rechenarbeit geeignet konstruierter Mehrgitteralgorithmen im vielen Fällen vergleichsweise niedrig ist und nur linear mit der Anzahl der Unbekannten im zu lösenden linearen Gleichungssystem wächst. Die asymptotische Optimalität der Mehrgitterverfahren zeigt deren niedrige Rechenintensität (d.h. deren niedriges Verhältnis von Gleitpunktoperationen pro Datenzugriff) auf und unterstreicht folglich die besondere dringende Notwendigkeit hoher Datenlokalität.

Als zweites präsentieren und untersuchen wir Ansätze zur Optimierung der Lokalität zellulärer Automaten, welche in erster Linie verwendet werden, um das zeitabhängige Verhalten komplexer Systeme in wissenschaftlichen und technischen Disziplinen zu studieren. Als Beispiel zellulärer Automaten mit hoher praktischer Relevanz verwenden wir die Lattice-Boltzmann-Methode, die insbesondere im Bereich der numerischen Strömungsmechanik und in den Materialwissenschaften

zum Einsatz kommt.

Optimierungsansätze

Zum einen behandelt unsere Arbeit Optimierungen, welche — zumindest aus theoretischer Sicht — automatisch mittels eines Compilers durchgeführt werden könnten. Diese Techniken zielen auf Restrukturierungen der Datenanordnung sowie der Datenzugriffe ab, so dass sich eine verbesserte Wiederverwendung von Cache-Inhalten ergibt und größere Mengen an Daten aus den höhreren Schichten der Speicherhierarchie geladen werden können. Unbedingt anzumerken ist, dass diese Optimierungsansätze keinesfalls die numerischen Eigenschaften der betrachteten Algorithmen verändern. Vielmehr bezwecken sie die Optimierung der Ausführungsgeschwindigkeit der Anwendungsprogramme ab, die im Falle von iterativen linearen Lösern in MFLOPS (*engl.:* millions of floating-point operations per second) und im Falle der Lattice-Boltzmann-Methode in MLSUPS (*engl.:* millions of lattice site updates per second) gemessen werden kann.

Zum anderen befassen wir uns mit dem Entwurf inhärent Cache-effizienter numerischer Verfahren. In der Tat stellt die Entwicklung neuer effizienter Algorithmen ein vielfältiges aktuelles Forschungsgebiet dar. Inbesondere werden wir einen neuartigen Mehrgitteralgorithmus vorstellen, der auf der Theorie der volladaptiven Mehrgittermethode beruht. Dieser Algorithmus erfordert eine umfassende mathematische Analyse und resultiert nicht aus der Anwendung automatisierbarer Programmtransformationen. Unser Mehrgitterverfahren stellt eine Kombination aus Patch-orientierter Gitterverarbeitung sowie adaptiven Relaxationstechniken dar. Infolgedessen weist er ein hohes Potential an Datenlokalität auf und kann daher sehr Cache-effizient implementiert werden.

Ferner ergibt sich, dass unsere Patch-adaptive Mehrgittermethode infolge des zu Grunde liegenden adaptiven Ansatzes dem Prinzip der lokalen Relaxation folgt und daher im Falle operatorinduzierter Singularitäten oder im Falle von Störungen durch Randbedingungen robuster arbeitet als Standardmehrgitterverfahren. Es erfolgt eine ausführliche Beschreibung unseres Algorithmus und die Darstellung experimenteller Ergebnisse.

Das DiME-Projekt

Unsere Arbeit wurde durch die *Deutsche Forschungsgemeinschaft (DFG)* gefördert, Projektkennzeichen Ru 422/7–1,2,3,5. Sie beruht zum Teil auf der Kooperation mit Dr. Christian Weiß, einem ehemaligen Mitarbeiter des *Lehrstuhls für Rechnertechnik und Rechnerorganisation/Parallelrechnerarchitektur, Technische Universität München*, der seine Promotion Ende 2001 abschloss [Wei01]. Beide Dissertationen müssen notwendigerweise zusammen betrachtet werden, um zu einem vollständigen Bild unserer Forschungstätigkeit sowie der Vielfalt der erzielten Ergebnisse zu gelangen. Im Folgenden werden wir daher oftmals auf [Wei01] verweisen.

Wir entschieden uns zugunsten von

DiME — data-local iterative methods

als Name für unser Forschungsvorhaben. Dieser wurde seither beibehalten und wird auch weiterhin verwendet werden, obwohl er die zusätzliche Arbeit an Lokalitätsoptimierungen für zelluläre Automaten nicht umfasst.

Gliederung

Die vorliegende Dissertation gliedert sich wie folgt. In Kapitel 2 fassen wir diejenigen Aspekte aus dem Bereich der Rechnerarchitektur zusammen, die für unsere Arbeit relevant sind. Insbesondere

erläutern wir den Aufbau von Speicherhierarchien.

In Kapitel 3 beschreiben wir in Kürze die erste Klasse von Zielalgorithmen, auf die wir unsere Techniken anwenden: iterative Lösungsverfahren für große dünn besetzte Systeme linearer Gleichungen. Wir beschreiben sowohl elementare iterative Löser als auch Mehrgitteralgorithmen.

Kapitel 4 enthält eine Beschreibung der zweiten Klasse von Zielalgorithmen: zelluläre Automaten. Wir beschreiben den allgemeinen Formalismus zellulärer Automaten und konzentrieren uns im Anschluss daran auf die Lattice-Boltzmann-Methode als ein repräsentatives und praxisrelevantes Beispiel.

In den Kapiteln 5 und 6 erläutern wir eine Vielzahl an Ansätzen, um die Lokalität der Datenanordnungen und der Datenzugriffe zu erhöhen. Wir untersuchen die Anwendung dieser Techniken auf beide Algorithmenklassen aus den Kapiteln 3 und 4.

Eine Darstellung experimenteller Ergebnisse erfolgt in Kapitel 7. Diese Ergebnisse umfassen sowohl Cache-optimierte Implementierungen iterativer linearer Löser als auch der Lattice-Boltzmann-Methode. Es werden Implementierungen der Algorithmen in 2D und in 3D behandelt.

Kapitel 8 konzentriert sich auf den Entwurf inhärent Cache-effizienter Mehrgitteralgorithmen zur numerischen Lösung partieller Differentialgleichungen. Insbesondere erläutern wir unseren Patch-adaptiven Mehrgitteralgorithmus und stellen experimentelle numerische Resultate vor.

Eine Übersicht ausgewählter verwandter Forschungsaktivitäten findet sich in Kapitel 9. Wir stellen allgemeine Ansätze der Effizienzoptimierung numerischer Software vor, Techniken für gitterbasierte Berechnungen eingeschlossen. Wir erläutern Lokalitätsoptimierungen für Algorithmen der numerischen linearen Algebra für voll besetzte Matrizen und geben eine Einführung in unsere Cache-optimierte Mehrgitterbibliothek *DiMEPACK*. In Anschluss daran erörtern wir unterschiedliche Lokalitätsmetriken und untersuchen als Beispiel die Anwendung einer dieser Metriken, um die Auswirkung einer Cache-basierten Programmtransformation zu quantifizieren. Schließlich stellen wir einen Ansatz zur automatisierten Integration bibliotheksspezifischer Quellcodetransformationen vor.

Abschließende Folgerungen aus unserer Arbeit werden in Kapitel 10 gezogen. Insbesondere beziehen wir Beispiele praktischer Anwendungen unserer Arbeit mit ein. Des Weiteren schlagen wir zukünftige Forschungsarbeiten im Bereich der architekturorientierten Optimierung numerischer Algorithmen vor.

Appendix C

Curriculum Vitae

General information

Date/place of birth:	June 29, 1972, Fürth, Germany
Office address:	System Simulation Group (Informatik 10), Department of Computer Science, Friedrich-Alexander University Erlangen-Nuremberg (FAU), Cauerstraße 6, D-91058 Erlangen, Germany
Email:	*markus.kowarschik@cs.fau.de*
WWW:	*http://www10.informatik.uni-erlangen.de/∼markus*

Education

12/1998–06/2004	PhD student, System Simulation Group (Informatik 10), Department of Computer Science, University of Erlangen-Nuremberg, Germany, 06/18/2004: graduation as *Doktor-Ingenieur (Dr.-Ing.)*
04/1998–11/1998	PhD student, Numerical Analysis Group, Department of Mathematics, University of Augsburg, Germany
11/1991–03/1998	Student, University of Erlangen-Nuremberg, Germany, major: computer science, minor: mathematics, 03/10/1998: graduation as *Diplom-Informatiker (Dipl.-Inf.)*
06/1991	*Allgemeine Hochschulreife (Abitur)*

Employment

05/2001–09/2001	Research assistant, Center for Applied Scientific Computing (CASC),
07/2002–10/2002	Lawrence Livermore National Laboratory (LLNL), Livermore, CA, USA
12/1998–	Research assistant, System Simulation Group (Informatik 10), Department of Computer Science, University of Erlangen-Nuremberg, Germany
04/1998–11/1998	Research assistant, Numerical Analysis Group, Department of Mathematics, University of Augsburg, Germany

General References

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 3. edition, 1999. `http://www.netlib.org/lapack/lug`.

[ABD+97] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 1–14, St. Malo, France, 1997.

[ABDP81] R.E. Alcouffe, A. Brandt, J.E. Dendy, and J.W. Painter. The Multi-Grid Method for the Diffusion Equation with Strongly Discontinuous Coefficients. *SIAM J. Sci. Stat. Comput.*, 2(4):430–454, 1981.

[AGGW02] B.S. Andersen, J.A. Gunnels, F. Gustavson, and J. Waśniewski. A Recursive Formulation of the Inversion of Symmetric Positive Definite Matrices in Packed Storage Data Format. In *Proc. of the 6th Int. Conf. on Applied Parallel Computing (PARA 2002)*, volume 2367 of *Lecture Notes in Computer Science (LNCS)*, pages 287–296, Espoo, Finland, 2002. Springer.

[Ahl66] L.V. Ahlfors. *Complex Analysis*. McGraw-Hill, 2. edition, 1966.

[AK01] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, San Francisco, California, USA, 2001.

[AMD03] AMD, Inc. *Software Optimization Guide for AMD Athlon 64 and AMD Opteron Processors*, 2003. Document Number: 25112.

[AMP00] N. Ahmed, N. Mateev, and K. Pingali. Tiling Imperfectly-Nested Loop Nests. In *Proc. of the ACM/IEEE Supercomputing Conf. (SC00)*, Dallas, Texas, USA, 2000.

[BACD97] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *Proc. of the Int. Conf. on Supercomputing (ICS'97)*, Vienna, Austria, 1997.

[Bäh02a] H. Bähring. *Mikrorechner-Technik, Band I.* Springer, 3. edition, 2002.

[Bäh02b] H. Bähring. *Mikrorechner-Technik, Band II.* Springer, 3. edition, 2002.

[BBC+94] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[BBG+94] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *Proc. of the 2nd Annual Object-Oriented Numerics Conf. (OON-SKI '94)*, Sunriver, Oregon, USA, 1994.

[BCGN00] S.B. Baden, N.P. Chrisochoides, D.B. Gannon, and M.L. Norman, editors. *Structured Adaptive Mesh Refinement (SAMR) Grid Methods*, volume 117 of *The IMA Volumes in Mathematics and its Applications*. Springer, 2000.

[BDG+00] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. of High Performance Computing Applications*, 14(3):189–204, 2000.

[BDJ03] A.H. Baker, J.M. Dennis, and E.R. Jessup. An Efficient Block Variant of GMRES. Technical Report #CU-CS-957-03, University of Colorado, Dept. of Computer Science, Boulder, Colorado, USA, July 2003.

[BDQ98] F. Bassetti, K. Davis, and D. Quinlan. Temporal Locality Optimizations for Stencil Operations for Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures. In *Proc. of the 10th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'98)*, Las Vegas, Nevada, USA, 1998.

[Ber97] K. Bernert. τ-Extrapolation — Theoretical Foundation, Numerical Experiment, and Application to Navier-Stokes Equations. *SIAM J. Sci. Comp.*, 18(2):460–478, 1997.

[BF01] R.L. Burden and J.D. Faires. *Numerical Analysis*. Brooks/Cole, 7. edition, 2001.

[BGS94] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[BHM00] W.L. Briggs, V.E. Henson, and S.F. McCormick. *A Multigrid Tutorial*. SIAM, 2. edition, 2000.

[BHQ98] D. Brown, W. Henshaw, and D. Quinlan. Overture: An Object-Oriented Framework for Solving Partial Differential Equations on Overlapping Grids. In *Proc. of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, Yorktown Heights, New York, USA, 1998. SIAM.

[BKT99] C. Becker, S. Kilian, and S. Turek. Consequences of Modern Hardware Design for Numerical Simulations and their Realization in FEAST. In *Proc. of the EuroPar-99 Conf.*, volume 1685 of *Lecture Notes in Computer Science (LNCS)*, pages 643–650. Springer, 1999.

[BM99] D. Bulka and D. Mayhew. *Efficient C++*. Addison-Wesley, 1999.

[BM00] R. Berrendorf and B. Mohr. PCL — The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors. Technical report, Research Center Jülich GmbH, Jülich, Germany, 2000. `http://www.fz-juelich.de/zam/PCL`.

[BP94] A. Berman and R.J. Plemmons. *Nonnegative Matrices in the Mathematical Sciences*. SIAM, 1994.

[Bra77] A. Brandt. Multi-level Adaptive Solutions to Boundary-Value Problems. *Math. Comput.*, 31:333–390, 1977.

[Bra84] A. Brandt. Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics. GMD Studie Nr. 85, St. Augustin, West Germany, 1984.

[Bra97] D. Braess. *Finite Elemente*. Springer, 2. edition, 1997.

[Cas00] G.C. Cascaval. *Compile-Time Performance Prediction of Scientific Programs*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, 2000.

[CB95] T.-F. Chen and J.-L. Baer. Effective Hardware Based Data Prefetching for High-Performance Processors. *IEEE Trans. on Computers*, 44(5):609–623, 1995.

[CD98a] S. Chen and G.D. Doolen. Lattice Boltzmann Method for Fluid Flows. *Ann. Rev. Fluid Mech.*, 30:329–364, 1998.

[CD98b] B. Chopard and M. Droz. *Cellular Automata Modeling of Physical Systems*. Cambridge University Press, 1998.

[CDE94] S. Chen, G.D. Doolen, and K.G. Eggert. Lattice-Boltzmann Fluid Dynamics. *Los Alamos Science*, 22:100–109, 1994.

[CHCM03] B. Chandramouli, W.C. Hsieh, J.B. Carter, and S.A. McKee. A Cost Model for Integrated Restructuring Optimizations. *J. of Instruction-Level Parallelism*, 5, 2003. `http://www.jilp.org/vol5`.

[CiS03] *IEEE/AIP Computing in Science and Engineering (CiSE)*, 5(1):18–38, 2003.

[Com99] Compaq Computer Corporation. *21264 Alpha Microprocessor Hardware Reference Manual*, 1999. Order Number: EC-RJRZA-TE.

[CP03] C. Cascaval and D.A. Padua. Estimating Cache Misses and Locality using Stack Distances. In *Proc. of the Int. Conf. on Supercomputing (ICS'03)*, San Francisco, California, USA, 2003.

[CPHL01] S. Chatterjee, E. Parker, P.J. Hanlon, and A.R. Lebeck. Exact Analysis of the Cache Behavior of Nested Loops. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'01)*, pages 286–297, Snowbird, Utah, USA, 2001.

[DAC00] A. Dupuis, P. Albuquerque, and B. Chopard. Impact de l'arrangement des données en mémoire dans un environnement parallèle dédié aux automates cellulaires. In *Rencontres Francophones du Parallélisme des Architectures et des Systèmes (RenPar'2000)*, Besançon, France, 2000.

[DBMS79] J. Dongarra, J. Bunch, C. Moler, and G.W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, Pennsylvania, USA, 1979.

[DCHD90] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[DD] C.C. Douglas and M.B. Douglas. MGNet Bibliography. Yale University, Department of Computer Science, New Haven, Conneticut, USA, 1991–. `http://www.mgnet.org/bib/mgnet.bib`.

[DHBW04] F. Deserno, G. Hager, F. Brechtefeld, and G. Wellein. Performance of Scientific Applications on Modern Supercomputers. In S. Wagner, W. Hanke, A. Bode, and F. Durst, editors, *High Performance Computing in Science and Engineering, Munich 2004*. Springer, 2004. To appear.

[Dig97] Digital Equipment Corporation. *Digital Semiconductor 21164 Alpha Microprocessor — Hardware Reference Manual*, 1997. Order Number: EC-QP99B-TE.

[DKP99] F. Dobrian, G. Kumfert, and A. Pothen. The Design of Sparse Direct Solvers Using Object-oriented Techniques. Technical Report 99–38, ICASE, NASA, Langley Research Center, Hampton, Virginia, September 1999.

[DLL01] N.P. Drakenberg, F. Lundevall, and B. Lisper. An Efficient Semi-Hierarchical Array Layout. In *Proc. of the 5th Ann. Workshop on Interaction between Compilers and Computer Architectures (INTERACT-5)*. Kluwer, 2001.

[Don04] J.J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, University of Tennessee, Computer Science Department, Knoxville, Tennessee, USA, April 2004. Regularly updated.

[Dou96] C.C. Douglas. Caching in With Multigrid Algorithms: Problems in Two Dimensions. *Parallel Algorithms and Applications*, 9:195–204, 1996.

[EJ98] H. Eisenmann and F.M. Johannes. Generic Global Placement and Floorplanning. In *Proc. of the 35th ACM/IEEE Design Automation Conf. (DAC'98)*, pages 269–274, San Francisco, California, USA, 1998.

[ES95] A. Eustace and A. Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proc. of the USENIX Technical Conf. on UNIX and Advanced Computing Systems*, pages 303–314, 1995.

[FJ98] M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, Washington, USA, 1998.

[Fla98] G.W. Flake. *The Computational Beauty of Nature*. MIT Press, 1998.

[FLPR99] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proc. of the 40th Symposium on Foundations of Computer Science*, pages 285–298, New York City, New York, USA, 1999.

[FS98] J. Fenlason and R. Stallman. *GNU gprof*. Free Software Foundation, Inc., Boston, Massachusetts, USA, 1998. `http://www.gnu.org`.

[GDN98] M. Griebel, T. Dornseifer, and T. Neunhoeffer. *Numerical Simulation in Fluid Dynamics*. SIAM, 1998.

[GGHvdG01] J.A. Gunnels, F.G. Gustavson, G.M. Henry, and R.A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 2001.

[GH01] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM, 2001.

[GKKS00] W.D. Gropp, D.K. Kaushik, D.E. Keyes, and B.F. Smith. Performance Modeling and Tuning of an Unstructured Mesh CFD Application. In *Proc. of the ACM/IEEE Supercomputing Conf. (SC00)*, Dallas, Texas, USA, 2000.

[GL98] G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins University Press, 3. edition, 1998.

[GL99] S. Guyer and C. Lin. An Annotation Language for Optimizing Software Libraries. In *Proc. of the 2nd Conf. on Domain-Specific Languages (DSL 99)*, pages 39–53, Austin, Texas, USA, 1999.

[GMM97] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. In *Proc. of the Int. Conf. on Supercomputing (ICS'97)*, pages 317–324, Vienna, Austria, 1997.

[GMM98] S. Ghosh, M. Martonosi, and S. Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 228–239, San Jose, California, USA, 1998.

[Gol91] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[Gre97] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. SIAM, 1997.

[Gri94a] M. Griebel. Multilevel Algorithms Considered as Iterative Methods on Semidefinite Systems. *SIAM J. Sci. Comput.*, 15(3):547–565, 1994.

[Gri94b] M. Griebel. *Multilevelmethoden als Iterationsverfahren über Erzeugendensystemen*. Teubner-Skripten zur Numerik. Teubner, 1994.

[Gut91] H.A. Gutowitz, editor. *Cellular Automata*. MIT Press, 1991.

[Hac85] W. Hackbusch. *Multigrid Methods and Applications*. Springer, 1985.

[Hac93] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*, volume 95 of *Applied Mathematical Sciences*. Springer, 1993.

[Hac96] W. Hackbusch. *Theorie und Numerik elliptischer Differentialgleichungen*. Teubner, 2. edition, 1996.

[Han98] J. Handy. *The Cache Memory Book*. Academic Press, 2. edition, 1998.

[HBR03] F. Hülsemann, B. Bergen, and U. Rüde. Hierarchical Hybrid Grids as Basis for Parallel Numerical Solution of PDE. In *Proc. of the EuroPar-03 Conf.*, volume 2790 of *Lecture Notes in Computer Science (LNCS)*, pages 840–843. Springer, 2003.

[Hig02] N.J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2. edition, 2002.

[HKN99] J. Harper, D. Kerbyson, and G. Nudd. Analytical Modeling of Set-Associative Cache Behavior. *IEEE Transactions on Computers*, 48(10):1009–1024, 1999.

[HKRG02] F. Hülsemann, P. Kipfer, U. Rüde, and G. Greiner. *gridlib:* Flexible and Efficient Grid Management for Simulation and Visualization. In P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, editors, *Proc. of the Int. Conf. on Computational Science (ICCS 2002), Part III*, volume 2331 of *LNCS*, pages 652–661, Amsterdam, The Netherlands, 2002. Springer.

[HL97] X. He and L.-S. Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, 56(6):6811–6817, 1997.

[HP03] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher, Inc., San Francisco, California, USA, 3. edition, 2003.

[HS02] T. Huckle and S. Schneider. *Numerik für Informatiker*. Springer, 2002.

[Hu00] J.J. Hu. *Cache Based Multigrid on Unstructured Grids in Two and Three Dimensions*. PhD thesis, Dept. of Mathematics, University of Kentucky, Lexington, Kentucky, USA, 2000.

[IBM01] IBM Corporation. *POWER4 System Microarchitecture*, 2001. Technical White Paper.

[Int02] Intel Corporation. *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, 1999–2002. Document Number: 248966–05.

[Int03] Intel Corporation. *Intel Itanium 2 Processor Reference Manual*, 2003. Document Number: 251110–002.

[IY01] E.-J. Im and K. Yelick. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In V.N. Alexandrov, J.J. Dongarra, B.A. Juliano, R.S. Renner, and C.J.K. Tan, editors, *Proc. of the Int. Conf. on Computational Science (ICCS 2001), Part I*, volume 2073 of *LNCS*, pages 127–136, San Francisco, California, USA, 2001. Springer.

[JL01] M. Jung and U. Langer. *Methode der finiten Elemente für Ingenieure*. Teubner, 2001.

[KA00] P. Knabner and L. Angermann. *Numerik partieller Differentialgleichungen — Eine anwendungsorientierte Einführung*. Springer, 2000.

[KBC+01] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries. *J. of Parallel and Distributed Computing*, 61(12):1803–1826, 2001.

[Key00] D.E. Keyes. Trends in Algorithms for Nonuniform Applications on Hierarchical Distributed Architectures. In M.D. Salas and W.K. Anderson, editors, *Proc. of the 1998 ICASE/LaRC Workshop on Computational Aerosciences for the 21st Century*, pages 103–137, Hampton, Virginia, USA, 2000. Kluwer.

[Key02] D.E. Keyes. Domain Decomposition Methods in the Mainstream of Computational Science. In I. Herrera, D.E. Keyes, O.B. Widlund, and R. Yates, editors, *Proc. of the 14th Int. Conf. on Domain Decomposition Methods*, pages 79–93, Cocoyoc, Mexico, 2002. UNAM Press.

[Kil02] S. Kilian. *SCARC. Ein verallgemeinertes Gebietszerlegungs-/Mehrgitterkonzept auf Parallelrechnern.* Logos, Berlin, 2002. Dissertation.

[Kum03] P. Kumar. Cache Oblivious Algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science (LNCS)*, pages 193–212. Springer, 2003.

[Löt96] H. Lötzbeyer. Parallele adaptive Mehrgitterverfahren: Ein objektorientierter Ansatz auf semistrukturierten Gittern. Forschungs- und Lehreinheit V, Institut für Informatik, TU Munich, Germany, December 1996. Diplomarbeit.

[LR97] H. Lötzbeyer and U. Rüde. Patch-Adaptive Multilevel Iteration. *BIT*, 37(3):739–758, 1997.

[LRB01] W.-F. Lin, S.K. Reinhardt, and D. Burger. Designing a Modern Memory Hierarchy with Hardware Prefetching. *IEEE Transactions on Computers*, 50(11):1202–1218, 2001.

[LRW91] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. of the Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Palo Alto, California, USA, 1991.

[MMC96] B. Mohr, A. Malony, and J. Cuny. TAU. In G. Wilson, editor, *Parallel Programming using C++*. MIT Press, 1996.

[MP99] V. Menon and K. Pingali. High-Level Semantic Optimization of Numerical Codes. In *Proc. of the Int. Conf. on Supercomputing (ICS'99)*, Rhodes, Greece, 1999.

[MRF+00] S.S. Mukherjee, S.K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M.D. Hill, J.R. Larus, and D.A. Wood. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 8(4):12–20, 2000.

[MSY02] R. Mei, W. Shyy, and D. Yu. Lattice Boltzmann Method for 3-D Flows with Curved Boundary. Technical Report 2002–17, ICASE, NASA, Langley Research Center, Hampton, Virginia, June 2002.

[Muc97] S.S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, San Francisco, California, USA, 1997.

[MV03] M. Mohr and B. Vanrumste. Comparing Iterative Solvers for Linear Systems associated with the Finite Difference Discretisation of the Forward Problem in Electroencephalographic Source Analysis. *Mathematical and Biological Engineering and Computing*, 41(1):75–84, 2003.

[NAW+96] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, 1996.

[NGDLPJ96] J.J. Navarro, E. Garcia-Diego, J.-L. Larriba-Pey, and T. Juan. Block Algorithms for Sparse Matrix Computations on High Performance Workstations. In *Proc. of the Int. Conf. on Supercomputing (ICS'96)*, pages 301–308, Philadelphia, Pennsylvania, USA, 1996.

[NS03] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Proc. of the 3rd Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, 2003.

[OJ04] B. Obermeier and F.M. Johannes. Temperature-Aware Global Placement. In *Proc. of the Asia South Pacific Design Automation Conf. (ASP-DAC 2004)*, volume 1, pages 143–148, Yokohama, Japan, 2004.

[PK94] S. Palacharla and R.E. Kessler. Evaluaing Stream Buffers as a Secondary Cache Replacement. In *Proc. of the 21st Int. Symposium on Computer Architecture (ISCA)*, pages 24–33, Chicago, Illinois, USA, 1994. IEEE Computer Society Press.

[PMHC03] V.K. Pingali, S.A. McKee, W.C. Hsieh, and J.B. Carter. Restructuring Computations for Temporal Data Cache Locality. *Int. J. of Parallel Programming*, 31(4):306–338, 2003.

[PSX+04] M. Püschel, B. Singer, J. Xiong, J.M.F. Moura, J. Johnson, D. Padua, M.M. Veloso, and R.W. Johnson. SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *J. of High Performance Computing and Applications*, 18(1):21–45, 2004. Special Issue: Automatic Performance Tuning.

[QMPS02] D. Quinlan, B. Miller, B. Philip, and M. Schordan. Treating a User-Defined Parallel Library as a Domain-Specific Language. In *Proc. of the 16th Int. Parallel and Distributed Processing Symposium (IPDPS 2002)*, pages 105–114, Ft. Lauderdale, Florida, USA, 2002. IEEE Computer Society Press.

[QSMK04] D. Quinlan, M. Schordan, B. Miller, and M. Kowarschik. Parallel Object-Oriented Framework Optimization. *Concurrency and Computation: Practice and Experience*, 16(2–3):293–302, 2004. Special Issue: Compilers for Parallel Computers.

[QSPK03] D. Quinlan, M. Schordan, B. Philip, and M. Kowarschik. The Specification of Source-to-Source Transformations for the Compile-Time Optimization of Parallel Object-Oriented Scientific Applications. In H.G. Dietz, editor, *Proc. of the 14th Workshop on Languages and Compilers for Parallel Computing (LCPC 2001)*, volume 2624 of *Lecture Notes in Computer Science (LNCS)*, pages 570–578, Cumberland Falls, Kentucky, USA, 2003. Springer.

[QSYdS03] D. Quinlan, M. Schordan, Q. Yi, and B.R. de Supinski. A C++ Infrastructure for Automatic Introduction and Translation of OpenMP Directives. In M.J. Voss, editor, *Proc. of the Int. Workshop on OpenMP Applications and Tools (WOMPAT 2003)*, number 2716 in Lecture Notes in Computer Science (LNCS), pages 13–25, Toronto, Canada, 2003. Springer.

[Qui00] D. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[QV99] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Numerical Mathematics and Scientific Computation. Oxford University Press, 1999.

[Riv01] G. Rivera. *Compiler Optimizations for Avoiding Cache Conflict Misses*. PhD thesis, Dept. of Computer Science, University of Maryland, College Park, Maryland, USA, 2001.

[RT98] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Montreal, Canada, 1998.

[RT00] G. Rivera and C.-W. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proc. of the ACM/IEEE Supercomputing Conf. (SC00)*, Dallas, Texas, USA, 2000.

[Rüd93a] U. Rüde. Fully Adaptive Multigrid Methods. *SIAM J. Num. Anal.*, 30(1):230–248, 1993.

[Rüd93b] U. Rüde. *Mathematical and Computational Techniques for Multilevel Adaptive Methods*, volume 13 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, Pennsylvania, USA, 1993.

[SB00] J. Stoer and R. Bulirsch. *Numerische Mathematik 2*. Springer, 4. edition, 2000.

[SC01] S. Sellappa and S. Chatterjee. Cache-Efficient Multigrid Algorithms. In V.N. Alexandrov, J.J. Dongarra, B.A. Juliano, R.S. Renner, and C.J.K. Tan, editors, *Proc. of the Int. Conf. on Computational Science (ICCS 2001), Part I*, volume 2073 of *LNCS*, pages 107–116, San Francisco, California, USA, 2001. Springer.

[Sch91] H.R. Schwarz. *Methode der finiten Elemente*. Teubner, 3. edition, 1991.

[Sch97] H.R. Schwarz. *Numerische Mathematik*. Teubner, 4. edition, 1997.

[She94] J.R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Unpublished draft, ed. $1\frac{1}{4}$, School of Computer Science, Carnegie Mellon University, see also `http://www.cs.berkeley.edu/~jrs`, 1994.

[Slo99] A. Slowik. *Volume Driven Selection of Loop and Data Transformations for Cache-Coherent Parallel Processors*. PhD thesis, Institut für Informatik, Universität Paderborn, Germany, Paderborn, Germany, 1999.

[Smi82] A.J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[SQ03] M. Schordan and D. Quinlan. A Source-To-Source Architecture for User-Defined Optimizations. In L. Böszörményi and P. Schojer, editors, *Proc. of the Joint Modular Languages Conf. (JMLC 2003)*, number 2789 in Lecture Notes in Computer Science (LNCS), pages 214–223, Klagenfurt, Austria, 2003. Springer.

[ST82] K. Stüben and U. Trottenberg. Multigrid Methods: Fundamental Algorithms, Model Problem Analysis, and Applications. In W. Hackbusch and U. Trottenberg, editors, *Multigrid Methods*, volume 960 of *Lecture Notes in Mathematics (LNM)*, pages 1–176. Springer, 1982.

[Ste94] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[Sto99] J. Stoer. *Numerische Mathematik 1*. Springer, 8. edition, 1999.

[Stü01] K. Stüben. An Introduction to Algebraic Multigrid. In U. Trottenberg, C.W. Oosterlee, and A. Schüller, editors, *Multigrid*, pages 413–532. Academic Press, 2001. Appendix A.

[Tal00] D. Talia. Cellular Processing Tools for High-Performance Simulation. *IEEE Computer*, 33(9):44–52, 2000.

[TGJ93] O. Temam, E. Granston, and W. Jalby. To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Proc. of the ACM/IEEE Supercomputing Conf. (SC93)*, Portland, Oregon, USA, 1993.

[TM87] T. Toffoli and N. Margolus. *Cellular Automata Machines*. MIT Press, 1987.

[Tol97] S. Toledo. Locality of Reference in LU Decomposition with Partial Pivoting. *SIAM J. on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[TOS01] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.

[Tse97] C.-W. Tseng. Data layout optimizations for high-performance architectures. Technical Report CS-TR-3818, Dept. of Computer Science, University of Maryland, College Park, Maryland, USA, Feb 1997.

[Ueb97a] C.W. Ueberhuber. *Numerical Computation 1*. Springer, 1997.

[Ueb97b] C.W. Ueberhuber. *Numerical Computation 2*. Springer, 1997.

[Var62] R.S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, 1962.

[VL00] S.P. Vanderwiel and D.J. Lilja. Data Prefetching Mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[VPVM98] G. Vahala, P. Pavlo, L. Vahala, and N.S. Martys. Thermal Lattice-Boltzmann Models (TLBM) for Compressible Flows. *Int. J. of Modern Physics C*, 9(8):1247–1261, 1998.

[WBR+03] Z. Wang, D. Burger, S.K. Reinhardt, K.S. McKinley, and C.C. Weems. Guided Region Prefetching: A Cooperative Hardware/Software Approach. In *Proc. of the 30th Int. Symposium on Computer Architecture (ISCA)*, pages 388–398, San Diego, California, USA, 2003.

[WD98] R.C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proc. of the ACM/IEEE Supercomputing Conf. (SC98)*, Orlando, Florida, USA, 1998.

[Wes92] P. Wesseling. *An Introduction to Multigrid Methods*. John Wiley & Sons, 1992.

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

[WL91] M.E. Wolf and M.S. Lam. A Data Locality Optimizing Algorithm. In *Proc. of the SIGPLAN'91 Symp. on Programming Language Design and Implementation*, volume 26 of *SIGPLAN Notices*, pages 33–44, Toronto, Canada, 1991.

[WM95] W.A. Wulf and S.A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[Wol94] S. Wolfram. *Cellular Automata and Complexity.* Addison-Wesley, 1994.

[Wol96] M.J. Wolfe. *High-Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, California, USA, 1996.

[WPD01] R.C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.

[Yav96] I. Yavneh. On Red-Black SOR Smoothing in Multigrid. *SIAM J. Sci. Comp.*, 17(1):180–192, 1996.

[You71] D.M. Young. *Iterative Solution of Large Linear Systems.* Academic Press, 1971.

DiME Project — References

[DHH+00a] C.C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. 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. Weiß. Portable Memory Hierarchy Techniques For PDE Solvers: Part II. *Siam News*, 33(6), July 2000.

[DHI+00] C.C. Douglas, J. Hu, M. Iskandarani, M. Kowarschik, U. Rüde, and C. Weiß. Maximizing Cache Memory Usage for Multigrid Algorithms. In Z. Chen, R.E. Ewing, and Z.-C. Shi, editors, *Proc. of the Int. Workshop on Computational Physics: Fluid Flow and Transport in Porous Media*, Lecture Notes in Physics (LNP), Beijing, China, 2000. Springer.

[DHK+00a] C.C. Douglas, J. Hu, W. Karl, M. Kowarschik, U. Rüde, and C. Weiß. Fixed and Adaptive Cache Aware Algorithms for Multigrid Methods. In E. Dick, K. Riemslagh, and J. Vierendeels, editors, *Multigrid Methods VI — Proc. of the 6th European Multigrid Conf.*, volume 14 of *Lecture Notes in Computational Science and Engineering (LNCSE)*, pages 87–93, Gent, Belgium, 2000. Springer.

[DHK+00b] C.C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transactions on Numerical Analysis (ETNA)*, 10:21–40, 2000.

[KKRW01] W. Karl, M. Kowarschik, U. Rüde, and C. Weiß. DiMEPACK — A Cache-Aware Multigrid Library: User Manual. Technical Report 01–1, Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Erlangen, Germany, March 2001.

[KRTW02] M. Kowarschik, U. Rüde, N. Thürey, and C. Weiß. Performance Optimization of 3D Multigrid on Hierarchical Memory Architectures. In *Proc. of the 6th Int. Conf. on Applied Parallel Computing (PARA 2002)*, volume 2367 of *Lecture Notes in Computer Science (LNCS)*, pages 307–316, Espoo, Finland, 2002. Springer.

[KRWK00] M. Kowarschik, U. Rüde, C. Weiß, and W. Karl. Cache-Aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions. *Computing*, 64:381–399, 2000.

[KW01] M. Kowarschik and C. Weiß. DiMEPACK — A Cache-Optimized Multigrid Library. In H.R. Arabnia, editor, *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, volume I, Las Vegas, Nevada, USA, 2001. CSREA, CSREA Press.

[KW02] M. Kowarschik and C. Weiß. Cache Performance Tuning of Numerically Intensive Codes. Technical Report 02–2, Lehrstuhl für Informatik 10 (Systemsimulation), University of Erlangen-Nuremberg, Germany, September 2002.

[KW03] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science (LNCS)*, pages 213–232. Springer, 2003.

[KWR02] M. Kowarschik, C. Weiß, and U. Rüde. Data Layout Optimizations for Variable Coefficient Multigrid. In P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, editors, *Proc. of the 2002 Int. Conf. on Computational Science (ICCS 2002), Part III*, volume 2331 of *Lecture Notes in Computer Science (LNCS)*, pages 642–651, Amsterdam, The Netherlands, 2002. Springer.

[PKW+03a] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde. Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes. *Parallel Processing Letters*, 13(4):549–560, 2003.

[PKW+03b] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde. Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes in 2D and 3D. Technical Report 03–8, Lehrstuhl für Informatik 10 (Systemsimulation), University of Erlangen-Nuremberg, Germany, July 2003.

[Rüd03] U. Rüde. A Measure for a Code's Temporal Locality, 2003. Private communication.

[Wei01] C. Weiß. *Data Locality Optimizations for Multigrid Methods on Structured Grids*. PhD thesis, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany, December 2001.

[WKKR99] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proc. of the ACM/IEEE Supercomputing Conf. (SC99)*, Portland, Oregon, USA, 1999.

[WKT04] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *Proc. of the 2004 Int. Conf. on Computational Science (ICCS 2004)*, Lecture Notes in Computer Science (LNCS), Kraków, Poland, 2004. Springer. To appear.

[WPKR03a] J. Wilke, T. Pohl, M. Kowarschik, and U. Rüde. Cache Performance Optimizations for Parallel Lattice Boltzmann Codes. In *Proc. of the EuroPar-03 Conf.*, volume 2790 of *Lecture Notes in Computer Science (LNCS)*, pages 441–450. Springer, 2003.

[WPKR03b] J. Wilke, T. Pohl, M. Kowarschik, and U. Rüde. Cache Performance Optimizations for Parallel Lattice Boltzmann Codes in 2D. Technical Report 03–3, Lehrstuhl für Informatik 10 (Systemsimulation), University of Erlangen-Nuremberg, Germany, July 2003.

DiME Project — Student Theses

[Chr03] I. Christadler. Patch-Adaptive Mehrgitterverfahren für partielle Differentialgleichungen. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, December 2003. Studienarbeit.

[Dau01] V. Daum. Runtime-Adaptivity Techniques for Multigrid Methods. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, August 2001. Studienarbeit.

[Igl03] K. Iglberger. Cache Optimizations for the Lattice Boltzmann Method in 3D. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, September 2003. Bachelor thesis.

[Pfä01] H. Pfänder. Cache-optimierte Mehrgitterverfahren mit variablen Koeffizienten auf strukturierten Gittern. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, January 2001. Diplomarbeit.

[Thü02] N. Thürey. Cache Optimizations for Multigrid in 3D. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, June 2002. Studienarbeit.

[WA99] H. Wörndl-Aichriedler. Adaptive Mehrgitterverfahren in Raum und Zeit. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, December 1999. Diplomarbeit.

[Wil03] J. Wilke. Cache Optimizations for the Lattice Boltzmann Method in 2D. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, February 2003. Studienarbeit.

[Zet00] M. Zetlmeisl. Performance Optimization Of Numerically Intensive Codes — A Case Study From Biomedical Engineering. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, November 2000. Studienarbeit.