Performance Optimization of 3D Multigrid on Hierarchical Memory Architectures*

Markus Kowarschik[1], Ulrich Rüde[1], Nils Thürey[1], and Christian Weiß[2]

[1] Lehrstuhl für Systemsimulation (Informatik 10)
Institut für Informatik
Friedrich–Alexander–Universität Erlangen–Nürnberg, Germany
{Markus.Kowarschik, Ulrich.Ruede, Nils.Thuerey}@cs.fau.de

[2] Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Fakultät für Informatik
Technische Universität München, Germany
Christian.Weiss@cs.tum.edu

Abstract. Today's computer architectures employ fast cache memories in order to hide both the low main memory bandwidth and the latency of main memory accesses, which is slow in contrast to the floating–point performance of the CPUs. Efficient program execution can only be achieved, if the codes respect the hierarchical memory design. Iterative methods for linear systems of equations are characterized by successive sweeps over data sets, which are much too large to fit in cache. Standard implementations of these methods thus do not perform efficiently on cache–based machines. In this paper we present techniques to enhance the cache utilization of multigrid methods on regular mesh structures in 3D as well as various performance results. Most of these techniques extend our previous work on 2D problems.

1 Introduction

The speed of computer processors has been increasing and will even continue to increase much faster than the speed of memory components. Hence, current memory chips based on DRAM technology cannot provide the data to the CPUs as fast as necessary. This memory bottleneck often results in significant idle periods of the processors and thus in very poor code performance — in terms of MFLOPS — compared to the theoretically available peak performances. In order to mitigate this effect, today's computer architectures use cache memories that store data frequently used by the CPU. Caches are usually based on SRAM chips that, on the one hand, are much faster than DRAM components, but, on the other hand, have comparatively small capacities, for both technical and economical reasons. Most of today's RISC–based workstations even use several levels of caches. One to three levels are common [7].

* This research is being supported in part by the *Deutsche Forschungsgemeinschaft* (German Science Foundation), projects Ru 422/7–1,2,3.

Efficient execution can only be achieved if the hierarchical structure of the memory subsystem — including main memory, caches, and the CPU registers — is respected by the code; i.e., if the code exhibits both *temporal locality* as well as *spatial locality*[3]. Unfortunately, current compilers cannot introduce highly sophisticated optimizing code transformations automatically. Much of this effort is therefore left to the programmer [10].

The second motivation for our research is based on theoretical results concerning the computational costs of numerical algorithms. Due to their asymptotically linear complexity, multigrid methods are among the most efficient methods for the solution of large systems of linear equations [14]. Such problems, for example, arise in the context of the numerical solution of partial differential equations, using approximations based on finite differences, finite elements, or finite volumes. Multigrid methods belong to the class of iterative algorithms; i.e., the underlying data sets are traversed successively until some convergence criteria are fulfilled and the iteration process terminates. From this point of view multigrid codes exhibit a high degree of temporal locality.

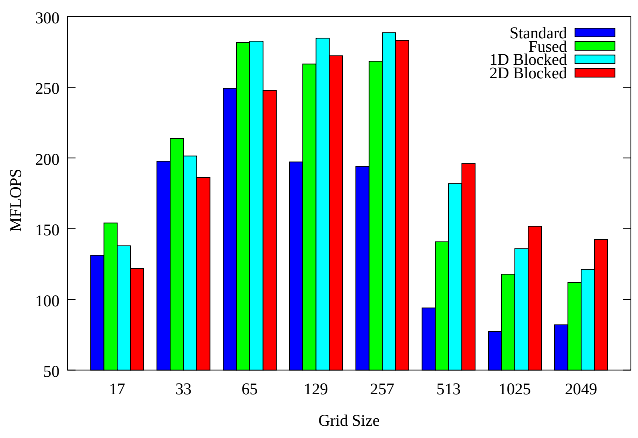

Fig. 1. *DiMEPACK* multigrid performance with different smoother optimizations.

However, current problems in science and engineering are based on data sets requiring memory capacities that easily exceed today's typical cache sizes. Hence, our research has focused on the investigation of techniques to optimize the cache performance of multigrid codes. Figure 1 shows speedups of factors up to more than 3 that are achieved on a Digital PWS 500au machine by using our 2D multigrid library *DiMEPACK* [8]. These enhancements are based on various *data layout and data access optimizations*, like array padding, loop fusion, as well

[3] *Temporal locality* means that data which have been accessed recently will be accessed again in the near future, *spatial locality* means that data which will be accessed next are located close in memory to the data which have been accessed recently [18].

as different loop blocking approaches. Detailed descriptions and further results for the 2D case can be found in [15, 16]. We presently focus on 3D problems.

While our efforts concentrate on algorithms for structured grids (see also [2]), other research activities address memory hierarchy optimizations for irregular mesh structures [4, 6] and a variety of other numerical algorithms; e.g., FFT [5]. Techniques for elementary stencil–based computations in 3D have recently been presented [12, 13].

This paper is structured as follows. Section 2 contains the model problem description and derives upper performance limits based on the underlying CPU architecture. Sections 3 and 4 describe our data layout optimization techniques and our optimizing data access transformations, respectively. Eventually, we draw several conclusions in Section 5.

2 Problem Description and Upper Performance Limits

In order to demonstrate the importance of data locality optimizations for 3D multigrid methods we will start with an analysis of the runtime behavior of a standard multigrid code. The code is based on a 7–point finite difference discretization of the differential operator. We discretize the continuous operator on each grid level anew such that the band structure of the matrices does not change from level to level. Our code employs a Gauss–Seidel smoother with a red/black ordering of the unknowns, full weighting as restriction operator, and it prolongates the corrections to the finer grids using tri–linear interpolation. Throughout the whole paper we use the scalar elliptic equation $\nabla \cdot (a \nabla u) = f$ with Dirichlet boundary conditions on the unit cube as our model problem.

The runtime behavior of the multigrid code on a Digital PWS 500au is summarized in Table 1. For all grid sizes, the performance is far away from the theoretical peak performance of 1 GFLOPS. Furthermore, the performance for the two largest problems is significantly lower than the performance of the multigrid code using smaller data sets. To detect why those performance drops occur, the program was profiled using *DCPI* [1]. 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), translation lookaside buffer misses (TLB), branch mispredictions (Branch), and register dependences (Depend).

For all grid sizes, instruction cache misses and branch mispredictions do not represent a significant performance bottleneck. For the smaller grids, register dependences are the limiting factor. However, with growing grid size, the memory behavior of the code dominates its runtime. Thus, for the two largest grid sizes with a memory consumption of about 35 MB and 290 MB, respectively, data cache miss stalls account for more than 60 per cent of all CPU cycles. In contrast to the 2D case, where TLB misses do not play an important role, they cause a significant number of idle cycles in 3D.

In order to get an impression of how fast multigrid codes can be executed on RISC–based microprocessors in general, we consider the instruction mix of a

Grid Size	MFLOPS	% of cycles used 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 1. Runtime behavior of a standard 3D multigrid code.

Grid Size	% of instructions executed as				
	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 2. Instruction mix of a standard 3D multigrid execution.

standard 3D multigrid code using different grid sizes. Our results are summarized in Table 2. For the larger grids, by far the most instructions are load as well as floating–point (FP) instructions, followed by integer operations. Most RISC architectures — including the Digital PWS 500au — execute data load/store operations within their integer units, which therefore have to process both integer and load/store operations. Thus, for a 129^3 grid the integer units have to process more than 60 per cent of all instructions.

In the case of the Digital PWS 500au, the ratio of integer units to FP units is $1 : 1$. Thus, if we assume an ideal case where no stalls of any kind occur, and where one integer and one FP instruction 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 FP instructions. Consequently, this limits the achievable FP performance to 50 per cent of the peak performance.

Although this observation is idealized, it gives an upper bound for the maximum performance of 3D multigrid methods. Higher performance can only be achieved if the ratio of FP instructions to load/store instructions changes. The following data locality optimizations are based on data dependence preserving loop transformations and data layout optimizations that in general do not reduce the number of load/store instructions[4].

[4] As a side effect, some of the optimizations might enable the compiler to reduce the number of load/store instructions [15].

3 Data Layout Optimizations

3.1 Cache–Aware Data Structures

In our case, each linear equation is characterized by nine FP values; the value of the corresponding unknown, the right-hand side, and the coefficients of the 7–point stencil. There is a variety of data storage schemes that can be used to store the linear system corresponding to each grid level [3, 9].

In [9] we have presented and investigated three different data layouts for the 2D case where a 5–point stencil is involved; *equation–oriented*, *band–wise*, and *access–oriented*. Each of these storage schemes can be easily extended to 3D problems. Our experiments have shown that the relative performance differences in 3D are comparable to the situation in 2D. We thus focus on the access–oriented data layout and refer to [9] for further details. This storage scheme uses two arrays; one array for the solution vector and one array of records each of which stores the right–hand side as well as the seven matrix entries corresponding to a single linear equation.

3.2 Array Padding

Array padding is a technique for eliminating *cache conflict misses* [13]. Conflict misses occur if the *associativity* of the cache [7] is not large enough to prevent frequently used cache lines from mutually evicting each other from the cache. The idea is to increase the array sizes by introducing elements that are never accessed. Consequently, the relative distances in memory and thus the mapping of the array elements to the cache locations are changed. The paddings to be inserted depend on the cache characteristics as well as on the array dimensions.

Several approaches to determining suitable array paddings are based on analytical cache models and heuristics concerning both program and cache behavior [12, 13]. However, other researchers argue that these models do not represent the system architecture in sufficient detail and thus only yield suboptimal paddings. They therefore propose array paddings based on searching the parameter space exhaustively [17]. While, in general, our research covers both alternatives we only focus on the second approach here.

Array padding techniques for 3D arrays typically enlarge the dimensions corresponding to the leading index and to the middle index. If the programming language standard prescribes *column major ordering* (e.g., in the case of FORTRAN77) the application of array padding looks as follows. The array declaration `double precision u(n1,n2,n3)` is replaced by the array declaration `double precision u(n1+pad1,n2+pad2,n3)`, where `pad1` and `pad2` are suitably chosen padding constants. `pad1` is introduced to avoid conflicts between array elements located in the same plane, whereas the purpose of `pad2` is to eliminate conflicts between neighboring elements in adjacent planes.

In contrast, we propose a non–standard padding approach for 3D arrays, which is characterized by less memory overhead than the standard one. Our technique introduces a relatively small number of *inter–plane* padding elements.

This number is not necessarily a multiple of the first array dimension, which obviously is always true for the conservative padding approach presented in the preceeding example.

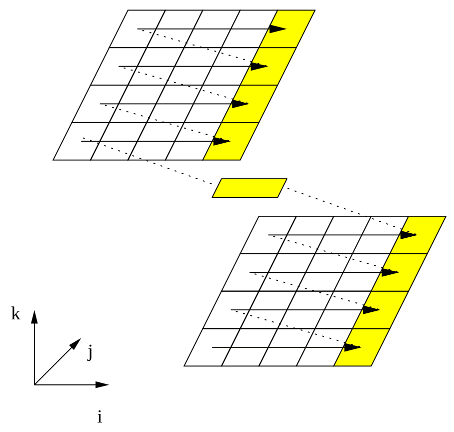

Fig. 2. Non–standard padding for 3D arrays, `i` is the leading, `k` is the trailing index.

Figure 2 illustrates how two adjacent planes of a 3D array are mapped into the address space as soon as our non–standard inter–plane padding is introduced. The shaded areas represent the padding.

The purpose of inter–plane padding is to avoid cache conflicts between neighboring elements located in adjacent planes. Hence, the use of inter–plane padding corresponds to the introduction of the constant `pad2` in the previous example. Our experiments reveal similar performance gains for both padding approaches.

4 Data Access Optimizations

Like data layout optimizations, data access optimizations also increase the locality behavior of the code. The core idea is that, by reducing the number of data accesses between two subsequent references to the same memory location, the data is more likely to still reside in cache or even in a CPU register.

We have shown in [15] that in 2D the most time–consuming part of a multigrid code is the smoother. This is also true in 3D. Profiling experiments confirm that, in the case of a V(2,2) multigrid cycle and for all problem sizes under consideration, the relaxation consumes more than 75% of the total execution time. The following optimizations thus address the implementation of the red–black Gauss–Seidel algorithm. It is important to point out that our techniques optimize the order of data accesses, while the numbering of the unknowns remains unchanged. Consequently, the convergence behavior of the method does not change and our optimized codes yield identical numerical results.

Since our Gauss–Seidel smoother is based on a red/black ordering of the unknowns and implements a 7–point stencil, the red nodes can be relaxed in any order. This is also true for the black nodes. This means that, in particular, the order of each loop nest can be changed without influencing the numerical results. For our standard implementation we use the most cache–aware loop order; the innermost loop accesses the array of the unknowns with unit stride. The performance of our standard code is relatively good for small problems, but decreases enormously as soon as the problem size exceeds the cache capacity.

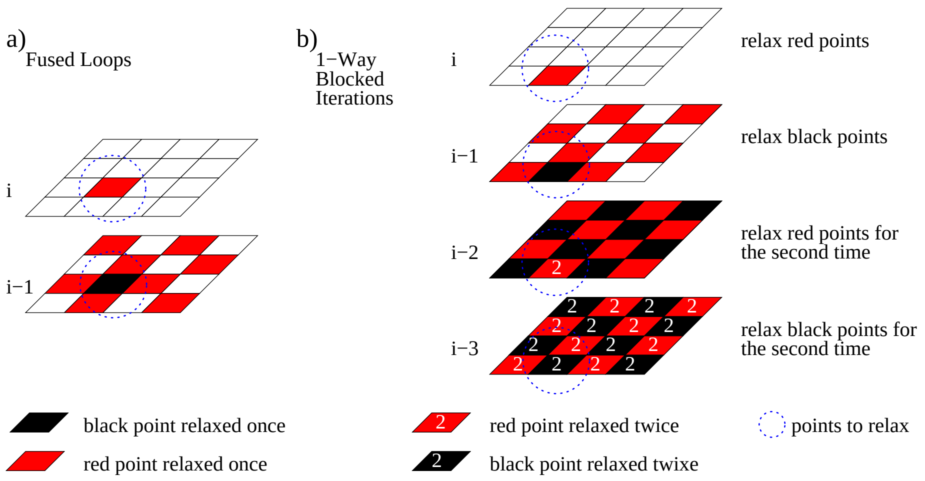

Fig. 3. a) Fused loops: the red point in plane i and afterwards the black point below in plane $i-1$ are relaxed together. b) Blocked iterations: two blocked iterations are illustrated, red and black points in different planes are updated in one step.

Analogously to the 2D case, our first optimization is to avoid the two traversals of the data set — one for the red nodes and a second one for the black nodes — by *fusing* the two loops into a single one. This can be done by relaxing a red point and immediately afterwards the black point below, see Figure 3a. If the cache can hold four complete planes of the data, all subsequent accesses can now be satisfied from it. Care has to be taken at the upper and lower boundaries.

The idea of reducing the number of traversals of the data set can be extended by *blocking* the iteration loop, see Figure 3b. A red point and its black neighboring point are relaxed in planes i and $i-1$, respectively. After that, the red point below in plane $i-2$ can be relaxed for the second time, likewise the black point in plane $i-3$. If two iterations are blocked the relaxation will then continue with the next red point in plane i. We use the term *1–way blocking* to refer to this technique since it involves one loop (the iteration loop) to be blocked. Note that, for N iterations in the standard red–black Gauss–Seidel version, $2N$ sweeps are necessary. By blocking the iterations loop by K, only N/K sweeps remain. In order that this technique works efficiently, the cache needs to hold $2K + 2$ adjacent planes.

Apparently, this is not possible for larger problem sizes. We therefore propose to partition the individual planes into rectangular tiles and to apply the previously explained technique in a plane–wise manner. If the tile size (TI, TJ) is chosen suitably, a sufficient number of tiles can be kept in cache. Since this technique requires three loops to be blocked (the iterations loop and two loops along array dimensions), we call this approach a *3–way blocking technique*. For the stencil described in Section 2 this would require that roughly the data for $(TI + 2) * (TJ + 2) * (2K + 2)$ points fit in cache.

In Figure 4 we present the MFLOPS rates that our 3D multigrid codes achieve on two different architectures[5]. Especially for the larger problems, the application of our techniques almost doubles the performance of the multigrid codes on both machines. For the smaller grids, however, the speedups differ. On the A21164–based Digital PWS 500au, our fusion technique yields the best performance due to the increasing loop overhead of the 1–way and the 3–way blocking approaches. This effect, however, can be compensated by the enhanced hardware features of the A21264–based Compaq XP 1000; e.g., out–of–order execution.

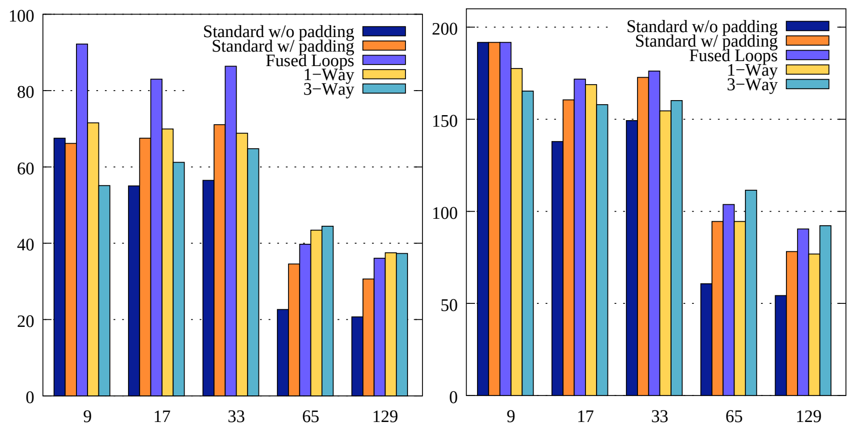

Fig. 4. MFLOPS rates of the optimized multigrid codes for two different architectures; left: A21164–based Digital PWS 500au, right: A21264–based Compaq XP 1000.

Apparently, the 3–way blocking technique does not perform much faster than the 1–way blocking technique on both machines. A more detailed profiling analysis on the Digital PWS 500au using DCPI [1] indicates the problem. While, in

[5] Our experiments were carried out on a Digital PWS 500au (A21164, 500 MHz, Compaq Tru64 UNIX V4.0D, DIGITAL Fortran V5.1) and on a Compaq XP 1000 (A21264, 500 MHz, Compaq Tru64 UNIX V4.0E, DIGITAL Fortran V5.2). On both platforms the compilers were configured to perform a large set of compiler optimizations.

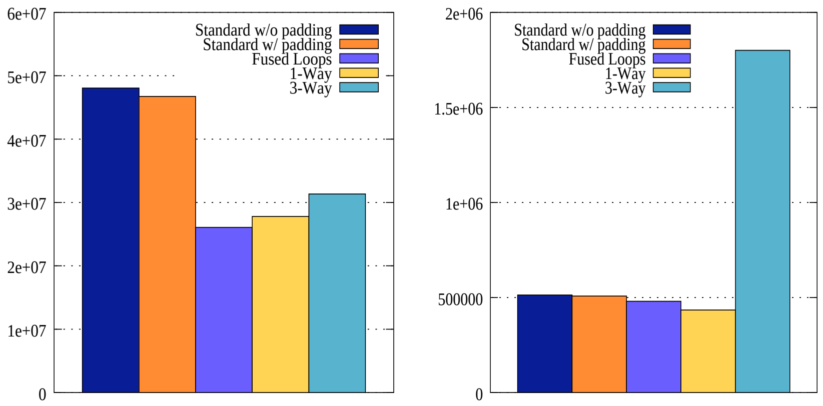

Fig. 5. L3 cache misses (left) and TLB misses (right) on a Digital PWS 500au for different code versions, problem size 129^3, five V(2,2) cycles.

particular, the number of L3 misses can almost be reduced by a factor of 2, the number of TLB misses grows by a factor of approximately 3.5, thus spoiling any positive data cache effects, see Figure 5.

5 Conclusions

We have presented techniques to enhance the cache performance of 3D multigrid codes. Most of our techniques carry over to a variety of iterative schemes based on regular mesh structures. Our practical experiments reveal that significant speedups can be achieved by applying suitable data layout transformations and data access optimizations. However, the performance gains are not as impressive as in the 2D case, see [15] for extensive 2D performance studies. One reason for this is the higher impact of the limited TLB capacity. Hence, it is inevitable to consider TLB behavior when tailoring numerical codes for hierarchical memory designs. Future work will further investigate these TLB effects and focus on techniques to overcome them; e.g., data copying.

Our current research activities focus on the investigation of analytical array padding approaches for variable–coefficient problems. These approaches are extensions of the techniques presented in [12]. Furthermore, we are presently developing new patch–based multilevel algorithms which are characterized by an inherent (i.e., algorithmic) locality behavior [11]. In short, our research efforts aim at the combination of both algorithmic as well as cache–oriented locality optimizations.

References

1. 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 *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 1–14, St. Malo, France, October 1997.

2. F. Bassetti, K. Davis, and D. Quinlan. Temporal Locality Optimizations for Stencil Operations within Parallel Object–Oriented Scientific Frameworks on Cache–Based Architectures. In *Proc. of the International Conf. on Parallel and Distributed Computing and Systems*, pages 145–153, Las Vegas, Nevada, USA, October 1998.
3. C.C. Douglas. Caching in With Multigrid Algorithms: Problems in Two Dimensions. *Parallel Algorithms and Applications*, 9:195–204, 1996.
4. 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*, 10:21–40, February 2000.
5. M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'98)*, volume 3, pages 1381–1384, May 1998.
6. W.D. Gropp, D.K. Kaushik, D.E. Keyes, and B.F. Smith. High Performance Parallel Implicit CFD. *Parallel Computing*, 27(4):337–362, March 2001.
7. J. Handy. *The Cache Memory Book*. Academic Press, second edition, 1998.
8. M. Kowarschik and C. Weiß. DiMEPACK — A Cache–Optimized Multigrid Library. In H.R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, volume I, pages 425–430, Las Vegas, NV, USA, June 2001. CSREA Press.
9. M. Kowarschik, C. Weiß, and U. Rüde. Data Layout Optimizations for Variable Coefficient Multigrid. In *Proceedings of the 2002 International Conference on Computational Science*, Lecture Notes in Computer Science, Amsterdam, The Netherlands, April 2002. Springer. to appear.
10. D. Loshin. *Efficient Memory Programming*. McGraw–Hill, 1998.
11. H. Lötzbeyer and U. Rüde. Patch–Adaptive Multilevel Iteration. *BIT*, 37(3):739–758, 1997.
12. G. Rivera. *Compiler Optimizations for Avoiding Cache Conflict Misses*. PhD thesis, Dept. of Computer Science, University of Maryland, College Park, MD, USA, 2001.
13. G. Rivera and C.-W. Tseng. Tiling Optimizations for 3D Scientific Computation. In *Proceedings of the ACM/IEEE Supercomputing 2000 Conference (SC2000)*, Dallas, TX, USA, November 2000.
14. U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.
15. 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.
16. C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proceedings of the ACM/IEEE SC99 Conference*, Portland, Oregon, November 1999.
17. R.C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the International Conference on Supercomputing*, Orlando, Florida, USA, November 1998.
18. M.E. Wolf and M.S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN'91 Symposium on Programming Language Design and Implementation*, volume 26 of *SIGPLAN Notices*, pages 33–44, Toronto, Canada, June 1991.