Data Layout Optimizations for Variable Coefficient Multigrid*

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

[1] Lehrstuhl für Systemsimulation (Informatik 10), Institut für Informatik, Universität Erlangen–Nürnberg, Germany
[2] Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM), Fakultät für Informatik, Technische Universität München, Germany

Abstract. Efficient program execution can only be achieved if the codes respect the hierarchical memory design of the underlying architectures; programs must exploit caches to avoid high latencies involved with main memory accesses. However, iterative methods like multigrid are characterized by successive sweeps over data sets, which are commonly too large to fit in cache.
This paper is based on our previous work on data access transformations for multigrid methods for constant coefficient problems. However, the case of variable coefficients, which we consider here, requires more complex data structures.
We focus on data layout techniques to enhance the cache efficiency of multigrid codes for variable coefficient problems on regular meshes. We provide performance results which illustrate the effectiveness of our layout optimizations in conjunction with data access transformations.

1 Introduction

There is no doubt about the fact that the speed of computer processors has been increasing and will even continue to increase much faster than the speed of main memory components. As a general consequence, 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 compared to the theoretically available peak performances.

To mitigate this effect modern computer architectures use cache memories in order to store data that are frequently used by the CPU (one to three levels of cache are common). Caches are usually based on SRAM chips which, on the one hand, are much faster than DRAM components, but, on the other hand, have rather small capacities for both technical and economical reasons [7].

From a theoretical point of view multigrid methods are among the most efficient algorithms for the solution of large systems of linear equations. They

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

belong to the class of iterative schemes. This means that the underlying data set, which in general is very large, must be processed repeatedly. Efficient execution can only be achieved if the algorithm respects the hierarchical structure of the memory subsystem including main memory, caches and the processor registers, especially by the order of memory accesses [14]. Unfortunately, today's compilers are still far away from automatically applying cache optimizations to codes such complex as multigrid. Therefore much of this optimization effort is left to the programmer.

Semantics–maintaining cache optimization techniques for constant coefficient problems on structured grids have been studied extensively in our *DiME*[1] project [9, 16]. Our previous work primarily focuses on data access transformation which improve temporal locality. With multigrid methods for variable coefficient problems a reasonable layout of the data structures, which implies both high spatial locality and low cache interference, becomes more important. Thus this paper focuses on data layout optimization techniques for variable coefficient multigrid on structured meshes. We investigate and demonstrate the effectiveness of the data access transformations in conjunction with our data layout optimization techniques. Of course, it is not always appropriate or even possible to use such regular grids. Complex geometries, for instance, may require the use of irregular meshes. Thus our techniques must be seen as efficient building blocks which motivate the use of regular grid structures whenever this appears reasonable.

First considerations of data locality optimizations for iterative methods have been published by Douglas [3] and Rüde [14]. Their ideas initiated our *DiME* project [9, 16] as well as other research [1, 15]. All techniques are mainly based on data access transformation techniques like loop fusion and tiling for multigrid methods on structured grids. More recent work [4, 8] also focuses on techniques for multigrid on unstructured meshes. Keyes et al. have applied data layout optimization and data access transformation techniques to other iterative methods [6]. Genius et al. have proposed an automatable method to guide array merging for stencil–based codes based on a meeting graph method [5]. Tseng et al. have recently demonstrated how tile size and padding size selection can be automated for computational kernels in three dimensions [13].

This paper is organized as follows. In Section 2 we consider data structures and data layout strategies for stencil–based computations on arrays. Furthermore we explain array padding as an additional data layout optimization technique. Section 3 discusses performance results for data access optimizations in conjunction with various data layouts on several machines. Finally Section 4 summarizes our results and draws some final conclusions.

2 Data Layout Optimizations

As our optimization target we choose a multigrid V–cycle correction algorithm, which is based on a 5–point discretization of the differential operator, and assume Dirichlet boundaries. Consequently each inner node is connected to four

[1] **D**ata–local **i**terative **ME**thods for the efficient solution of PDEs

neighboring nodes. We use a red/black Gauss–Seidel smoother, full–weighting to restrict the fine–grid residuals, and linear interpolation to prolongate the coarse–grid corrections.

2.1 Data Storage Schemes

The linear system of equations is written as $Au = f$, the number of equations is denoted by n. The linear equation for a single inner grid point $i, 1 \leq i \leq n$ reads as: $so_i u_{so(i)} + we_i u_{we(i)} + ce_i u_i + ea_i u_{ea(i)} + no_i u_{no(i)} = f_i$. In the case of a constant coefficient problem an iterative method only needs to store five floating–point values besides the unknown vector u and the right–hand side f.

For variable coefficient problems, however, five coefficients must be stored for each grid point. Hence, the memory required for the coefficients outnumbers the storage requirements for the vectors u and f. There is a variety of data layouts for storing the unknown vector u, the right–hand side f, and the bands of the matrix A. In the following we will investigate three different schemes.

- *Equation–oriented storage scheme:* For each equation the solution, the right–hand side and the coefficients are stored adjacently as shown in Figure 1. This data layout is motivated by the structure of the linear equations.
- *Band–wise storage scheme:* The vectors u and f are kept in separate arrays. Furthermore, the bands of A are stored in separate arrays as well. This rather intuitive data layout is illustrated in Figure 2.
- *Access–oriented storage scheme:* The vector u is stored in a separate array. For each grid point i, the right–hand side f_i and the five corresponding coefficients so_i, we_i, ce_i, ea_i, and no_i are stored adjacently, as illustrated in Figure 3.

While the access–oriented storage scheme does not seem intuitive, it is motivated by the architecture of cache memories. Whenever an equation is being relaxed, its five coefficients and its right–hand side are needed. Therefore it is reasonable to lump these values in memory such that cache lines contain data which are needed simultaneously. This *array merging* technique [7] thus enhances spacial locality.

In the following, we will investigate the performance of a variable coefficient multigrid code which is written in C and uses double precision floating–point numbers. In our standard version one red/black Gauss–Seidel iteration is implemented as a first sweep over all red nodes and a second sweep over all black nodes. Cache–aware smoothing methods will be discussed in Section 3. Figure 4 shows the resulting MFLOPS rates for the three data layouts on different architectures. The finest grid comprises 1025 nodes in each dimension[2]. We use Poisson's equation as our model problem.

[2] Our experiments have been performed on a Compaq XP 1000 (A21264, 500 MHz, Compaq Tru64 UNIX V4.0E, Compaq cc V5.9), a Digital PWS 500au (A21164, 500 MHz, Compaq Tru64 UNIX V4.0D, Compaq cc V5.6), and a Linux PC (AMD Athlon, 700 MHz, gcc V2.35). On all platforms the compilers were configured to perform a large set of compiler optimizations.

It is obvious that the access–oriented storage scheme leads to the best performance on each platform. This validates our above considerations concerning its locality behavior. Moreover, except for the Compaq XP 1000, the equation–oriented technique yields higher execution speeds than the band–wise storage scheme as long as array padding is not introduced. This is due to the fact that the band–wise layout is highly sensitive to *cache thrashing*, see Section 2.2.

It is remarkable that, for example, the access–oriented data layout yields about 60 MFLOPS on the Compaq XP 1000 machine. This corresponds to 6% of the theoretically available peak performance of approximately 1 GFLOPS. The results for the A21164–based Digital PWS 500au and for the Athlon–based PC are even worse, since — according to the vendors — these three machines provide the same theoretical peak performances.

2.2 Array Padding

The performance of numerically intensive codes often suffers from *cache conflict misses*. These misses occur as soon as the associativity of the cache is not large enough. As a consequence, data that are frequently used may evict each other from the cache [12], causing cache thrashing. This effect is very likely in the case of stencil–based computations where the relative distances between array entries remain constant in the course of the passes through the data set. It is particularly severe as soon as the grid dimensions are chosen to be powers of 2, which is often the case for multigrid codes. In many cases *array padding* can help to avoid cache thrashing: the introduction of additional array entries, which are never accessed during the computation, changes the relative distances of the array elements and therefore eliminates cache conflict misses.

The automatic introduction of array padding to eliminate cache conflict misses is an essential part of today's compiler research [12]. However, current techniques to determine padding sizes are based on heuristics which do not lead to optimal results in many cases. It is thus a common approach to run large test suites in order to determine appropriate padding sizes [17].

Figure 5 shows the performance of a multigrid code based on band–wise data storage scheme for a variety of intra– and inter–array paddings on a Digital PWS 500au. The finest grid comprises 1025 nodes in each dimension. If no padding is applied (this situation corresponds to the origin $(0, 0)$ of this graph), poor performance results due to severe cache thrashing effects between the arrays holding bands of the matrix [11].

Our experiments have shown that the application of array padding hardly influences the execution times obtained for the equation–oriented storage scheme. This can be explained by the inherent inefficiency of this data layout: whenever an unknown is relaxed, the approximations corresponding to its four neighboring grid nodes are needed. Since, for each unknown, the approximative solution, the coefficients and the right–hand side are lumped in memory (Figure 1), most of the data which are loaded into the cache are not used immediately. This is particularly true for the coefficients and the right–hand side corresponding to the southern neighbor of the current grid point. It is likely that these data are

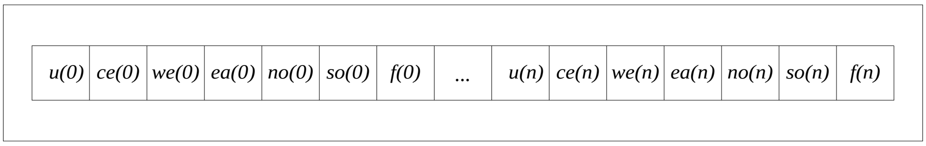

Fig. 1. Equation–oriented storage scheme.

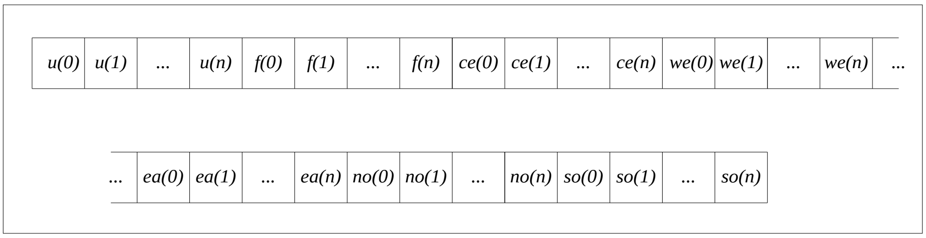

Fig. 2. Band–wise storage scheme.

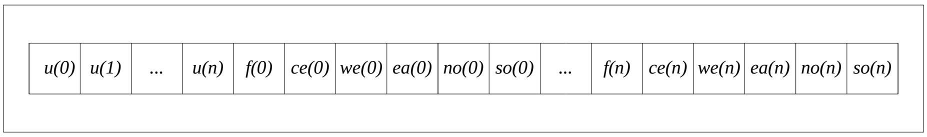

Fig. 3. Access–oriented storage scheme.

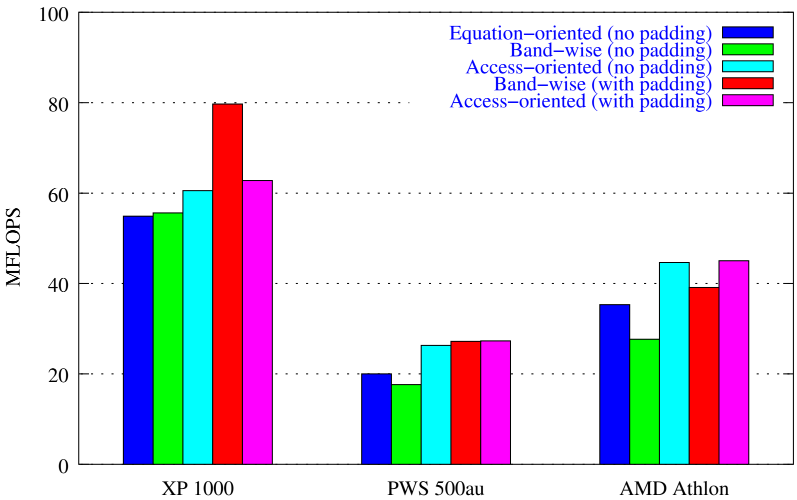

Fig. 4. CPU times for the multigrid codes based on different data layouts with and without array padding.

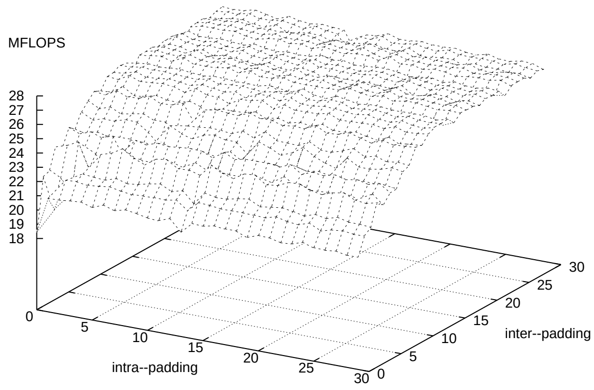

Fig. 5. MFLOPS rates for a multigrid code with different padding sizes using the band–wise storage scheme on a Digital PWS 500au .

evicted from the cache before they will be reused in the course of the next iteration. Consequently, the equation–oriented data layout poorly exploits spatial locality and will therefore no longer be considered here.

On all machines under consideration — the introduction of appropriate array paddings implies lower execution times if the band–wise storage scheme is used. The sensitivity of the code efficiency on the padding sizes mainly depends on the cache characteristics of the underlying machine; e.g., on the degrees of associativity. Detailed profiling experiments exhibit that, particularly for the Digital PWS 500au, the L1 miss rate and the L2 miss rate are reduced by more than 40% and 30%, respectively, as soon as padding is applied suitably to the band–wise data layout.

The third observation is that the performance of the multigrid code which employs the access–oriented storage scheme is always better or at least close to the performance for the band–wise data layout and, moreover, rather insensitive to array padding. Measurements using *PCL* [2] reveal that the cache miss rates almost remain constant. Therefore, as long as neither the programmer nor the compiler introduce array padding, this must be regarded as an advantage of the access–oriented storage scheme.

3 Data Access Optimizations

Data access transformations have been shown [9, 16] to be able to accelerate the red/black Gauss–Seidel smoother for constant coefficient problems by a multiple. Since the smoother is by far the most time–consuming part of a multigrid method this also leads to a significant speedup of the whole algorithm. The optimization techniques described extensively in [16] include the *fusion*, *1D blocking*, and *2D blocking* techniques. In the following, we will verify the effectiveness of the data access transformations in conjunction with our data layout optimization techniques.

Since all these techniques merely concern the implementation of the red/black Gauss–Seidel smoother, we only consider the performance of the smoothing routine in the following experiments. Besides, from here on, we use suitable array paddings for all our experiments.

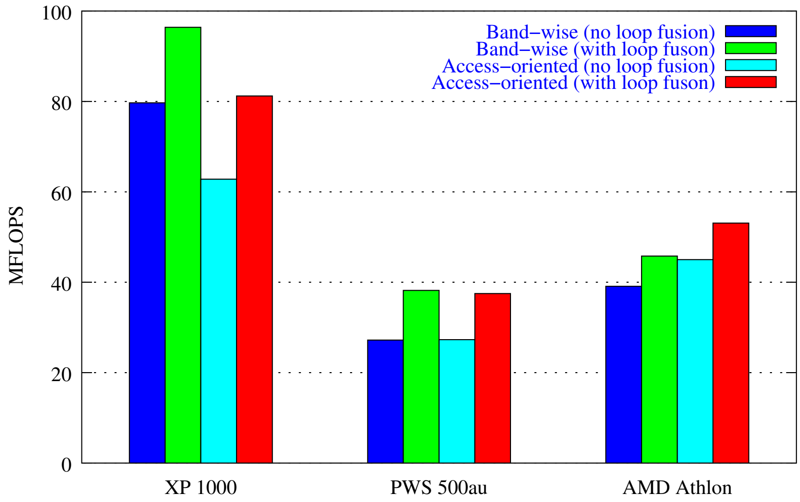

Fig. 6. MFLOPS rates for the smoothing routine based on different data layouts with and without loop fusion.

Figure 6 shows MFLOPS rates for the red/black Gauss–Seidel smoother on a square grid with 1025 nodes in each dimension. Again, we consider the efficiency of our codes on various platforms, with and without introducing the loop fusion technique. Both the band–wise storage scheme and the access–oriented data layout are taken into account. The efficiency for both Alpha–based machines still benefits from the application of loop fusion, whereas the performance gain on the Athlon–based PC is only marginal. This is due to the fact that the L2 cache of this processor has a capacity of 512 KB, which turns out to be too small to keep a sufficient number of neighboring grid lines, each of which containing 1025 nodes.

The same argument applies in the case of the 1D blocking. The application of the 1D blocking technique does not significantly enhance the performance of our smoothing routine further on the Athlon–based PC (Figure 7). However, both Alpha–based architectures have an additional off–chip cache of 4 MB, and, as a consequence, they benefit from blocking two ($m = 2$) or even four ($m = 4$) Gauss–Seidel iterations into one single pass through the grid.

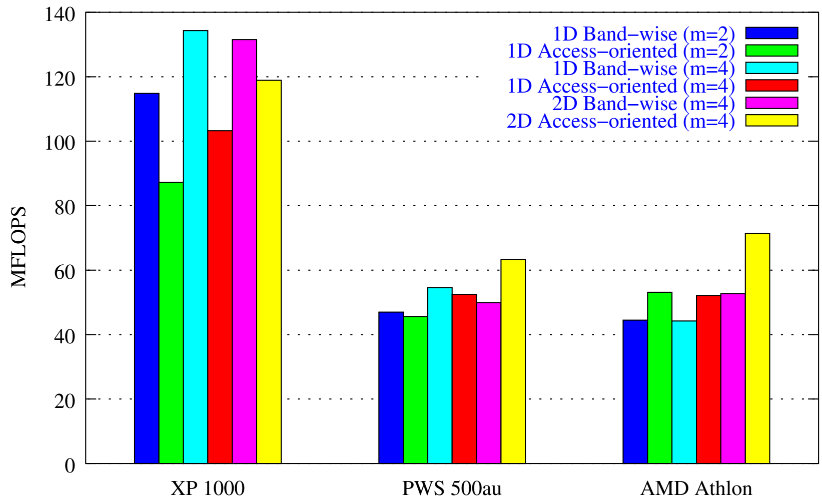

Fig. 7. MFLOPS rates for the smoothing routine based on different data layouts with 1D blocking ($m = 2, 4$) and 2D blocking ($m = 4$).

The situation, however, is different in the case of the 2D blocking technique. Figure 7 shows the performance after applying the 2D blocking technique to the red/black Gauss–Seidel smoother of our multigrid code. Four ($m = 4$) Gauss–Seidel iterations have been blocked into a single sweep over the grid, thus enhancing the reuse of cache contents and reducing the number of cache misses. The most important observation is that not only for both Alpha–based machines with the large off–chip caches, but also for the PC with only two levels of smaller caches, the MFLOPS rates can drastically be increased.

Consider for instance the MFLOPS rates for the AMD Athlon machine in Figure 6, which have been obtained by introducing the loop fusion technique. This comparison shows that, if the access–oriented storage scheme is used, the application of the 2D blocking technique can raise the MFLOPS rate by another 70%.

Varying the grid sizes reveals that, for smaller grids, 1D blocking leads to better performances than 2D blocking. The reason for this is that, if the grid lines are small enough, a sufficient number of them can be kept in cache, and 1D blocking causes efficient reuse of data in cache. If, however, the grid lines are getting larger, not enough of them can be stored in cache, and thus the additional overhead caused by the 2D blocking approach is over–compensated by the performance gain due to a higher cache reuse.

Figure 8 summarizes the influence of our optimizations on the cache behavior and the resulting execution times of our red/black Gauss–Seidel smoothers on the A21164–based Digital PWS 500au, again using a square grid with 1025 nodes in each dimension. The results for the optimization techniques loop fusion, 1D blocking and 2D blocking are based on the use of appropriate array paddings. It

	Standard	Loop Fusion	1D Blocking	2D Blocking
L1 misses	$3.7 \cdot 10^8$	76%	73%	78%
L2 misses	$1.5 \cdot 10^8$	87%	93%	41%
L3 misses	$7.5 \cdot 10^7$	52%	19%	16%
CPU time	20.0	13.3	10.0	8.0

Fig. 8. Summary of the numbers of L1, L2 and L3 cache misses and the CPU times in seconds for 40 iterations of the Gauss–Seidel smoothers on the Digital PWS 500au, the numbers of cache misses in the "Standard" column correspond to 100% each.

is apparent that especially the number of L3 misses can be drastically reduced, which is the main reason for the speedup factor of 2.5.

However, it must be mentioned that the speedups which can be achieved are not as significant as the impressive speedups which are obtained for constant coefficient codes, see e.g. [16]. This is due to the higher memory traffic required by variable coefficient codes. Nevertheless, our techniques yield speedup factors of 2 to 3.

4 Conclusions

In order to achieve efficient code execution it is inevitable to respect the hierarchical memory designs of today's computer architectures. We have presented optimization techniques which can enhance the performance of stencil–based computations on array data structures. Both data layout transformations and data access transformations can help to enhance the temporal and spatial locality of numerically intensive codes and thus their cache performance. We have shown that the choice of a suitable data layout — including the introduction of appropriate array padding — is crucial for efficient execution. This has been demonstrated for a variety of platforms.

Our research clearly illustrates the inherent performance penalties caused by the enormous gap between CPU speed — in terms of MFLOPS rates — and the speed of main memory components — in terms of access latency and memory bandwidth — and the resulting high potential for optimization.

Our experiments motivate the research on new numerical algorithms which can exploit deep memory hierarchies more efficiently than conventional iterative schemes. Therefore, our future work will focus on the development, performance analysis and optimization of patch–adaptive multigrid methods [10], which are characterized by a high inherent potential of data locality.

References

1. 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, Las Vegas, Nevada, USA, Oct. 1998, pp. 145–153.
2. R. Berrendorf, *PCL — The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 2.0)*, Forschungszentrum Juelich GmbH, Germany, `http://www.fz-juelich.de/zam/PCL`, Sept. 2000.
3. C. Douglas, *Caching in With Multigrid Algorithms: Problems in Two Dimensions*, Parallel Algorithms and Applications, 9 (1996), pp. 195–204.
4. 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 (2000), pp. 21–40.
5. D. Genius and S. Lelait, *A Case for Array Merging in Memory Hierarchies*, in Proceedings of the 9th Workshop on Compilers for Parallel Computers (CPC'01), Edinburgh, Scotland, June 2001.
6. W. Gropp, D. Kaushik, D. Keyes, and B. Smith, *High Performance Parallel Implicit CFD*, Parallel Computing, 27 (2001), pp. 337–362.
7. J. L. Hennessy and D. A. Patterson, *Computer Architecture — A Quantitative Approach*, Morgan Kaufmann Publishers, second ed., 1996.
8. J. Hu, *Cache Based Multigrid on Unstructured Grids in Two and Three Dimensions*, PhD thesis, Department of Mathematics, University of Kentucky, 2000.
9. M. Kowarschik, U. Rüde, C. Weiss, and W. Karl, *Cache–Aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions*, Computing, 64 (2000), pp. 381–399.
10. H. Lötzbeyer and U. Rüde, *Patch–Adaptive Multilevel Iteration*, BIT, 37 (1997), pp. 739–758.
11. H. Pfänder, *Cache–optimierte Mehrgitterverfahren mit variablen Koeffizienten auf strukturierten Gittern*, Master's thesis, Department of Computer Science, University of Erlangen–Nuremberg, Germany, 2000.
12. G. Rivera and C.-W. Tseng, *Data Transformations for Eliminating Conflict Misses*, in Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), Montreal, Canada, June 1998.
13. G. Rivera and C.-W. Tseng, *Tiling Optimizations for 3D Scientific Computation*, in Proceedings of the ACM/IEEE SC00 Conference, Dallas, Texas, USA, Nov. 2000.
14. U. Rüde, *Iterative Algorithms on High Performance Architectures*, in Proceedings of the EuroPar97 Conference, Lecture Notes in Computer Science, Springer, Aug. 1997, pp. 26–29.
15. S. Sellappa and S. Chatterjee, *Cache–Efficient Multigrid Algorithms*, in Proceedings of the 2001 International Conference on Computational Science (ICCS 2001), vol. 2073 and 2074 of Lecture Notes in Computer Science, San Francisco, California, USA, May 2001, Springer, pp. 107–116.
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, Nov. 1999.
17. R. C. Whaley and J. Dongarra, *Automatically Tuned Linear Algebra Software*, in Proceedings of the ACM/IEEE SC98 Conference, Orlando, Florida, USA, Nov. 1998.