Fixed and Adaptive Cache Aware Algorithms for Multigrid Methods*

Craig C. Douglas[1], Jonathan Hu[1], Wolfgang Karl[2], Markus Kowarschik[3], Ulrich Rüde[3], and Christian Weiß[2]

[1] University of Kentucky, Department of Mathematics, Lexington, KY, USA
[2] Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM), Technische Universität München, Germany
[3] Lehrstuhl für Systemsimulation (IMMD X), Universität Erlangen-Nürnberg, Germany

Abstract. Many current computer designs, including the node architecture of most parallel supercomputers, employ caches and a hierarchical memory structure. Hence, the speed of a multigrid code depends increasingly on how well the cache structure is exploited. Typical multigrid applications are running on data sets much too large to fit into any cache. Thus, applications should reuse copies of the data that is once brought into the cache as often as possible. In this paper, suitable fixed and adaptive blocking strategies for both structured and unstructured grids are introduced.

1 Introduction

At the present state of technology, main memory is very slow compared to the processing speed of the CPUs. Therefore, the cost of memory access is a serious bottleneck in the performance of many modern computers. Architectures are often comprised of caches and a hierarchical memory structure. Increasingly faster but also smaller memory units are employed to store the most frequently used data and as such speed up the overall computing time.

In general, efficient cache use requires the *locality* of memory accesses. It is expensive to bring data from the slower to the faster levels of the memory hierarchy, but once it is there, the reuse of the same data is much cheaper. Consequently, applications must be structured such that the *working set*, that is the most frequently accessed data, can fit in the fastest possible level of the memory hierarchy.

With such a design, the speed of a code (e.g., multigrid) depends on how well the cache structure is exploited, that is how frequent the access to cached data is relative to the number of all memory accesses. With the current transparent cache designs, a programmer can only indirectly influence which data is stored in the cache. Performance therefore depends on cleverly designed algorithms and data structures.

* This project is partially funded by the DFG Ru 422/7-1,2, NSF grants DMS-9707040 and ACR-9721388, and NATO grant CRG 971574.

In general, iterative methods successively perform global sweeps through their data structures and have a high potential for data reuse. The possible number of reuses is always at least as high as the number of iterations of the smoother or rougher plus the residual correction. Typical multigrid applications, however, are running on data sets much too large to fit into any cache. For a straightforward multigrid implementation, caches are therefore disappointingly ineffective. Most standard multigrid codes run only at a small fraction of the possible machine speed.

In this paper we demonstrate techniques how the data reuse within a multigrid algorithm can be improved by exploiting the locality of memory accesses with suitable blocking strategies for both structured and unstructured grids. The general idea is to block the grid points into subsets (or subdomains) and try to perform as much processing as possible within that block, before switching to the next one.

Clearly, this must be done carefully in order not to violate any data dependencies. Certain operations cannot be performed before neighboring blocks have been appropriately manipulated, so that quite involved strategies can result. Here we only focus on algorithms which are numerically identical (that is bitwise compatible) with standard multigrid methods. In terms of numerical performance criteria, like convergence, standard results apply to our algorithms. Still our algorithms are substantially faster than the corresponding standard algorithms, since the operations are reordered and can then be performed faster because there are fewer memory stalls.

Besides the fixed blocking strategy, we also introduce *adaptive blocking* where instead of a fixed set of unknowns, we use a *sliding* block or *active set* of unknowns that should be in cache and can be reused.

2 Structured Grids

The smoother or rougher is typically the most time consuming part of a multigrid method. To motivate cache optimization for multigrid methods we examine the runtime behavior of a two dimensional red-black Gauss-Seidel relaxation algorithm for a structured grid using a 5-point discretization of the Laplacian operator on a Digital PWS 500au. Table 1 summarizes the analysis with the profiling tool *DCPI* [1]. The result of the analysis is a breakdown of CPU cycles spent for execution (Exec), nops, and different kinds of stalls (see Table 1). Possible causes for stalls are data cache misses (Cache), data table lookaside buffer misses (DTB), branch mispredictions (Branch), and register dependencies (Depend). For the smaller grid sizes, the limiting factors are branch mispredictions and register dependencies. With growing grid sizes, however, the cache behavior of the algorithm becomes the dominating reason for the poor performance of the code. Thus, for the largest grids, data cache miss stalls account for more than 80% of all CPU cycles.

Data locality optimizations reorder the data accesses so that as few of them as possible are performed between any two data references to the same

Grid Size	MFLOPS	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
16	347.0	60.7	0.3	2.6	6.7	21.1	4.5
32	354.8	59.1	10.9	7.0	4.6	11.0	5.4
64	453.9	78.8	1.4	15.7	0.1	0.0	4.2
128	205.5	43.8	6.3	47.5	0.0	0.0	2.4
256	182.9	31.9	60.6	4.2	0.0	0.0	3.3
512	63.7	11.3	85.2	2.2	0.0	0.0	1.2
1024	58.8	10.5	85.9	2.4	0.0	0.0	1.1
2048	55.9	10.1	86.5	2.4	0.0	0.0	1.1

Table 1. Runtime behavior of standard red-black Gauss-Seidel.

memory location. With this, it is more likely that the data is not evicted from the cache and therefore can be loaded from one of the higher levels of the hierarchy. However, the new access order is only valid if data dependencies are still observed.

In the case of a 5-point red-black Gauss-Seidel method we can update the red nodes in any row and the black nodes in the row below in pairs without violating any data dependencies, instead of performing one global sweep through the whole grid updating all the red nodes and then another sweep updating all the black nodes. This is a *fusion technique*.

This technique applies only to one single red-black Gauss-Seidel sweep. If several successive iterations must be performed, the data in the cache is not yet reused from one sweep to the next, if the grid is too large to fit entirely in the cache. It is possible to update the red nodes in a line i for the second time, however, provided that all neighboring black nodes have been updated once. This is the case as soon as the black node in line $i+1$ directly above the red node has been touched once. As described before, this black node in turn can be updated as soon as the red node in line $i+2$ directly above it has been updated for the first time. Consequently, we can update the red nodes in rows $i+2$ and i and the black nodes in rows $i+1$ and $i-1$ in pairs. This *blocking technique* can be generalized to more than just two successive red-black Gauss-Seidel sweeps.

Both of these techniques require a certain number of rows to fit entirely in the cache. The larger grids, however, will not fit completely into higher levels of the memory hierarchy, in particular the registers and the L1 cache. A high utilization of the registers and the L1 cache, however, is crucial for the performance of any computationally intensive method. We therefore suggest a two dimensional blocking strategy [2]. The key idea for this technique is to move a small two dimensional block through the grid, updating all the nodes within the block. The block must be shaped as a parallelogram in order to obey all the data dependencies, and the update operations within the parallelogram are performed in a linewise manner from top to bottom.

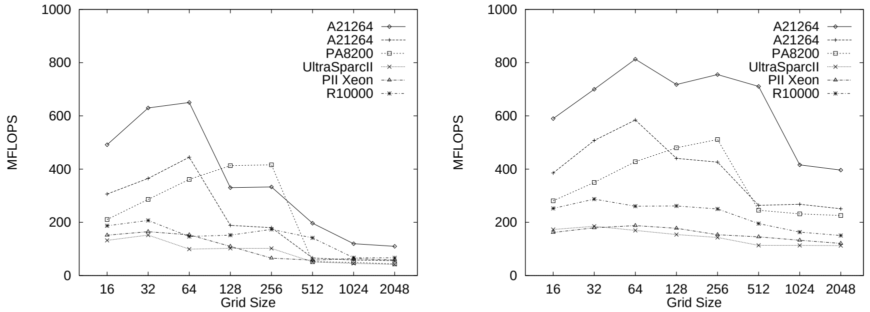

Fig. 1. MFLOPS for a 5-point standard (left side) and an optimized (right side) red-black Gauss-Seidel method on several platforms.

The principle of the technique as well as the fusion and blocking techniques are described in more detail in [6].

Figure 1 shows the performance of a standard red-black Gauss-Seidel implementation (left side) compared to the best possible performance obtained by the previously described optimizations (right side) on several platforms. Our program was compiled with native FORTRAN77 compilers and aggressive optimizations enabled. On the Intel machine we used *egcs* (V2.91.60). The platforms include an Intel PentiumII Xeon PC (450 MHz, 450 MFLOPS), a SUN Ultra 60 (296 MHz, 592 MFLOPS), a HP SPP2200 Convex Exemplar Node (200 MHz, 800 MFLOPS), a Compaq PWS 500au (500 MHz, 1 GFLOPS), and a Compaq XP1000 (500 MHz, 1 GFLOPS). Especially for the larger grids speedups of 2-5 can be observed.

3 Unstructured Grids

As in the structured case, we again are optimizing the smoother portion of the multigrid code. Our strategy involves several preprocessing steps: physical grid decomposition into *cache blocks*, renumbering of cache blocks, and reordering of operators.

After the preprocessing, we can perform cache aware Gauss-Seidel: update as much as possible on each cache block without referencing data from other cache blocks, calculate the residual wherever possible on the last Gauss-Seidel step, and revisit cache blocks as necessary to finish updating nodes on cache block boundaries.

The first step of our strategy is to decompose the grid on each multigrid level into *cache blocks*. A cache block is a connected set of grid nodes. A cache block should have the property that the corresponding matrix rows, unknowns, and right hand side values all fit into cache at the same time. Furthermore, the decomposition of the problem grid into cache blocks should also have the property that connections between blocks are minimized while the number of nodes in the interior is maximized. Many readily available load balancing packages for parallel computers are designed to do just this (we use METIS [5]).

Once a cache block is identified, we need to know how many relaxations are possible for each node without referencing another block. Within a cache block, the kth *subblock* consists of those nodes which can be updated at most k times without referencing nodes in other cache blocks. The *cache block boundary* $\partial\Omega_j$ is the set of nodes in cache block Ω_j which are adjacent to some node in cache block $\Omega_i, i \neq j$.

The number of relaxations possible on any node i in Ω_j is the length of the shortest path between i and any node in $\partial\Omega_j$, where the length of a path is the number of nodes in a path. The work required to find the distance of every node in Ω_j from $\partial\Omega_j$ is $\mathcal{O}(n)$. See [4] for a description of the algorithms.

We assume that the grid has been divided into k cache blocks and that within a block the numbering is contiguous. In general, let L_i^j denote those nodes in block j which are distance i from $\partial\Omega_j$. We renumber the nodes in Ω_j, beginning with subblock L_1^j and ending with L_l^j, where l is the number of subblocks in Ω_j. The result is a nodal ordering which is contiguous within blocks and subblocks. In the new ordering, nodes in Ω_j which are closer to $\partial\Omega_j$ will have a higher number than those which are further from $\partial\Omega_j$.

Once all matrix and grid operators have been reordered, the multigrid scheme can be applied. Assume that m smoothing steps are applied. On one grid level, within cache block Ω_j, all nodes receive one update. All nodes in subblocks $L_{m+1}^j, \ldots, L_2^j$ are updated a second time. All nodes in subblocks $L_{m+1}^j, \ldots, L_3^j$ are updated a third time. This proceeds until all nodes in L_m^j and L_{m+1}^j have been updated $m-1$ times. Finally, all nodes in L_{m+1}^j and L_m^j are updated once more, a partial residual is calculated in L_m^j, and the entire residual is calculated in L_{m+1}^j. Of course, Ω_j must be revisited to complete updates and residual calculations for nodes in $L_{m-1}^j, \ldots, L_1^j$.

A multigrid strategy also requires residual calculations. To maintain cache effects obtained during the smoothing step, the residual should also be calculated in a cache aware way during the last iteration of the smoothing.

An alternative to fixed cache block schemes is to reorder the matrices using a bandwidth reduction algorithm. Cache aware Gauss-Seidel can now be thought of as an active set of unknowns. An unknown remains in the active set until it is fully updated. This idea is motivated in [3].

Define the bandwidth B of a matrix $A = a(i,j)$ of order N to be $B = \min_{1 \leq i \leq N}\{\tau(i)\}$, where $\tau(i) = \max\{j - i : a(i,j) \neq 0, j > i\}$. An unknown i depends on unknowns j, where $j - i \leq B$ for $a(i,j) \neq 0$. For example, let α, β, and δ be sets of consecutive indices. All unknowns in $\alpha = \{1, \ldots, B\}$ can be updated for the nth time as soon as all unknowns in $\beta = \{B+1, B+2, \ldots, 2B\}$ have been updated $n-1$ times. In turn, all unknowns in β can be updated for the $(n-1)$st time as soon as all unknowns in $\delta = \{2B+1, \ldots, 3B\}$ have been updated $n-2$ times.

Let m be the number of Gauss-Seidel updates desired and s be a positive integer. In order to use a variable cache block smoothing scheme, the following must hold for $mB + s$ consecutive rows of A and a cache of size C:

$$C \geq M_u + M_r\{mB + s\} := C_{\min}(m), \tag{1}$$

where s is a positive integer, M_u is the memory required for $B(m+1)+s$ unknowns, and M_r is the memory required for one row of the matrix. M_r depends upon matrix storage implementation. In general, the residual can be completely calculated for the first s unknowns in cache and partially calculated for the next B unknowns. For simplicity, we assume that $s = B$, i.e., $C_{\min}(m) = M_u + M_r(m+1)B$.

In Table 2 the abbreviations GSS, GSI, GSAS, and CBGS stand for Gauss-Seidel with separate residual, integrated residual, variable cache blocks, and fixed cache blocks, respectively. Testing was done on one node of an HP SPP2200 with 200 MHz 8200 PA-RISC CPUs and on an SGI O2 with a 150 MHz IP32 R10000 CPU.

All tests solve a two dimensional linear elastic problem on a domain shaped like the state of Texas. The domain is discretized with linear triangular elements, and each node has two degrees of freedom. The northernmost horizontal border has zero Dirichlet boundary conditions at the corner points, and a force is applied at the southernmost tip in the downward (southern) direction. We emphasize that both schemes treat this problem as if it were variable coefficient, although it is not.

The variable cache block scheme seems to outperform the fixed cache block scheme on the SGI O2. The variable scheme also has less preprocessing steps to perform. This scheme is limited, however, by the bandwidth of the system matrix. If the bandwidth B is too large, the cache may be smaller than $C_{\min}(m)$. As a result, updates and residuals must be calculated using data which is not in cache.

Testing also indicates that $C_{\min}(m)$ is the best choice for the available cache size, rather than a larger size. Choosing the smallest allowable cache size may maximize the possibility that data is actually found in cache.

4 Conclusions

Implementing cache aware algorithms is a difficult and error prone task. In few application projects will there be the time to carefully hand tune algorithms in the style which we are proposing.

Making the optimized algorithms available as library routines is one of our present goals, but it must be clear that this can solve the problem only for a limited set of standard algorithms.

Variable cache block scheme on SGI O2				
smoother	# relaxations			
	2	3	4	5
GSS	4.19	5.53	6.87	8.19
GSI	4.30	5.65	6.99	8.33
GSAS	2.17	2.53	2.93	3.41
speedup	1.93	2.19	2.34	2.40

Variable cache block scheme on Exemplar SPP				
smoother	# relaxations			
	2	3	4	5
GSS	3.12	4.09	5.06	5.98
GSI	3.00	3.85	4.80	5.75
GSAS	1.54	1.79	2.09	2.42
speedup	1.95	2.15	2.30	2.38

Fixed cache block scheme on SGI O2				
smoother	# relaxations			
	2	3	4	5
GSS	4.16	5.51	7.07	8.12
GSI	4.30	5.61	7.01	8.28
CBGS	2.60	3.02	3.58	4.11
speedup	1.60	1.82	1.96	1.98

Fixed cache block scheme on Exemplar SPP				
smoother	# relaxations			
	2	3	4	5
GSS	3.23	4.02	4.19	6.12
GSI	3.08	3.99	4.96	5.88
CBGS	1.57	1.77	2.14	2.43
speedup	1.96	2.25	2.32	2.42

Table 2. CPU time (seconds) for 3-level multigrid V cycles using either variable or fixed cache block scheme.

To make the techniques applicable to a wider set of applications we are currently investigating techniques to automate the program transformations and to make the technology available either as preprocessing tools or as integrated compiler optimization techniques. For this purpose, it is a prerequisite that we do not change the algorithms, but only the order in which the data is accessed.

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, Oct. 1997.
2. C. C. Douglas. Caching in with multigrid algorithms: problems in two dimensions. *Paral. Alg. Appl.*, 9:195–204, 1996.
3. C. C. Douglas. Reusable cache memory object oriented multigrid algorithms. Preprint, 1999.
4. C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache optimization for structured and unstructured grid multigrid. *Electron. Trans. Numer. Anal.*, 9, 2000.
5. G. Karypis. METIS serial graph partitioning and matrix ordering software. In URL http://www-users.cs.umn.edu/~karypis/metis/metis/main.shtml.
6. C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Proceedings of the Supercomputing Conference*, Portland, Oregon, Nov. 1999.