

A Note on Cache Memory Methods for Multigrid in Three Dimensions

Craig C. Douglas and Dan T. Thorne

ABSTRACT. High performance computation occurs today only when the cache memory subsystem of a computer system is exploited. Peak speeds are unattainable for PDE solvers. A common misconception is that compilers will provide as high a percentage of the peak as is attainable.

Compilers should provide automatic tiling of regular grids for simple iterative algorithms. Language standards and the way most algorithms are implemented in code interfere with the automatic process. Hence, cache aware algorithms must to be developed and implemented instead.

Multigrid algorithms combine several numerical algorithms into a more complicated algorithm. In this paper, an algorithm for three dimensional elliptic boundary value problems is derived that allows for data to pass through cache exactly once per multigrid level during a V cycle before the level changes. This is optimal cache usage for large problems that do not fit entirely in cache.

1. Introduction

Multigrid methods are widely known as the fastest methods for solving elliptic partial differential equations. This belief was derived when computers were designed very differently than today. Accessing one word of data took a set amount of time due to computers having one level of memory.

Since the early 1980's, processors have sped up 5 times faster per year than memory. Multilevel memories, using *memory caches* were developed to compensate for the uneven speedups in the hardware. Essentially all computers today, from laptops to distributed memory supercomputers, use cache memories to keep the processors busy.

By the term cache, we mean a fast memory unit closely coupled to the processor [10, 12]. In the interesting cases, the cache is further divided into a great many *cache lines*, which hold copies of contiguous locations of main memory. The cache

1991 *Mathematics Subject Classification.* 68W10, 65Y05.

This research has been partially supported by gifts from Intel and Hewlett-Packard and grants from Sandia National Laboratories, NSF grants DMS-9707040, ACR-9721388, ACR-9814651, CCR-9902022, and CCR-9988165, National Computational Science Alliance grants OCE980001 and OCE010002 (utilizing the University of Illinois Origin 2000 and the University of New Mexico Los Lobos systems).

lines may hold data from quite separate parts of main memory. A good cache primer for solving PDEs can be found in [7, 8].

Tiling is the process of decomposing a computation into smaller blocks and doing all of the computing in each block one at a time. Tiling is an attractive method for improving data locality. In some cases, compilers can do this automatically. However, this is rarely the case for realistic scientific codes. In fact, even for simple examples, manual help from the programmers is, unfortunately, necessary [2].

Language standards interfere with compiler optimizations. Due to the requirements about loop variable values at any given moment in the computation, compilers are not allowed to fuse nested loops into a single loop. In part, it is due to coding styles that make very high level code optimization (nearly) impossible [13].

In this paper, we consider simple tensor product grids in three dimensions. We use both two and three dimensional concepts in our algorithms. Surprisingly, many cache problems that are solvable on two dimensional grids [4] are not completely solvable on a three dimensional grid [9].

Multigrid algorithms combine a number of operations in order to work. These include iterative methods (typically relaxation methods), residual computation, projection of residuals onto a coarser grid, and interpolation of corrections onto a finer grid. These are typically programmed as separate routines, which makes the components easy to replace and modify.

However, a number of components re-use data in a manner that is suitable for algorithms that are *cache aware*. Algorithms will be developed such that data passes through the memory cache once while computing on a given level before a level change.

This paper is concerned with algorithmic changes that are highly portable. Such techniques as loop unrolling, though mentioned, are not really stressed. The intention is that codes written using the algorithms in this paper will work well on anything from a PC to a high end RISC processor based supercomputer with only trivial tuning (one parameter). This means that we are not trying to get every last floating and fixed point operation out of a computer, just an integer factor speedup for a modest amount of work.

In §2, some introductory material on caches is presented. In §3, a once through the cache Gauss-Seidel is defined. In §4, multigrid is studied. In §5, performance is discussed and some conclusions are drawn.

2. Caches

A very good primer for how caches affect multigrid solvers is in [7, 8].

A computer's CPU performs the numerical and logical calculations when a program is executed. The data for the CPU is stored in main memory. When the CPU requires data that it does not already have, it makes a request to memory.

Many hardware and software strategies exist to reduce the time that a CPU is waiting for data. The most common hardware strategy is to divide computer memory into a hierarchy of different layers with the CPU linked directly to the highest level, or L1, cache. These layers are referred to as the cache or the memory subsystem [10, 12].

Caches are motivated by two principles of locality: temporal and spatial. The principle of temporal locality states that data required by the CPU now will also

be necessary again in the near future. The principle of spatial locality states that if specific data is required now, its neighboring data will be referenced soon.

The cache is itself a hierarchy of levels, called L1, L2, . . . , each with a different size and speed. One or two levels of cache is typical. The lower the L-number, the smaller the cache and the more expensive and faster it is.

The smallest block of data moved in and out of a cache is a cache line. A cache line holds data which is stored contiguously in main memory. A typical cache line is 32, 128, or 256 bytes in length, with 32 being most common.

If the data that the CPU requests is in a cache, it is called a cache hit. Otherwise it is a cache miss and data must be copied from a lower level (i.e., a slower) of memory into a cache. The hit rate is the ratio of cache hits to total memory requests. The miss rate is one minus the hit rate.

In designing algorithms to maximize cache hits, we must first decide which cache to optimize for. Many CPU's have a small L1 cache of only 8 – 96 KB, which is usually too small for 3D simulations involving coupled PDEs. Most computers have a larger L2 cache, which we tailor our algorithms for.

An exception is the HP PA series of CPU's, which have a large on-CPU L1 cache, but no L2 cache. Each of the PA-8500 and PA-8600 processors has a 1.5 MB L1 cache. On the PA-8700 processor is a 2.25 MB L1 cache.

Caches on any level may be unified or split. Unified caches mix both computer instructions and data. Split caches are two separate caches with instructions in one and data in the other. Split caches are superior to unified ones since the hit ratio tends to be much better for instructions than in unified caches.

In order to service CPU requests efficiently the memory subsystem must be able to determine whether requested data is present in a cache and where in the cache that data resides. There are two common methods used: direct mapping and k -way set associative.

The oldest method is the direct mapped cache. The location in cache is determined using the remainder of the (physical) main memory address divided by the cache size:

$$\text{cache address} = (\text{main memory address}) \bmod (\text{size of cache})$$

Several addresses in the cache are located in the same cache line. Hence, there is only one location in cache that a given main memory address can occupy.

Another hardware cache technique is k -way set associativity, which maps memory to k “ways,” or segments, inside a cache. A given memory address could be placed in any of the k segments of the cache. (A direct mapped cache corresponds to a 1-way set associative cache.) A policy for deciding in which “way” a piece of data should be stored is required for set associative caches. There are three common algorithms for choosing in which “way” a cache line should be placed: least recently used (LRU), least frequently used (LFU), and random replacement. These are called replacement policies since when a cache line is placed into cache it replaces a cache line that is already there.

One of the nastiest aspects of caches is known as thrashing. Two or more sets of data need to be used, but each map to the same cache lines. Hence, accessing the data always means a cache miss. The usual fix is to pad some part of the data in order to have different parts of the data map to different cache lines.

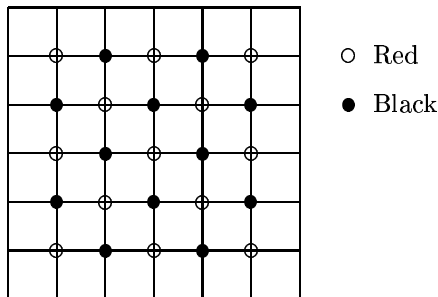


FIGURE 1. Simple 2D grid

3. Gauss-Seidel

Consider solving the set of linear equations

$$Au = f,$$

where $u \in \mathbb{R}^n$ and A is derived from discretizing an elliptic boundary problem. In the multigrid section, §4, we will add a subscript to the variables to indicate which level we are computing on.

Gauss-Seidel solves at each iteration

$$(L + D)u^{i+1} = f - Uu^i, \quad \text{where } A = D - L - U,$$

where D is diagonal and L and U are strictly lower and upper triangular and u^0 is a given initial guess.

Before transforming the standard Gauss-Seidel algorithms into cache aware versions, let us define two operations for updating the approximate solution on either one or two rows of a two dimensional grid:

$$\textit{Update}(\text{ row, color } [, \text{ direction}])$$

and

$$\textit{UpdateRedBlack}(\text{ row, } [, \text{ direction}])$$

We implicitly assume that both *Update* and *UpdateRedBlack* only compute on rows that actually exist.

The operation *Update* does a Gauss-Seidel update in each of the columns of a single row in the grid. The color is one of red, black, or all. The direction is optional and can be natural (the default) or reverse. Hence, symmetric Gauss-Seidel is easily implemented.

The operation *UpdateRedBlack* operates on all of the red points (x_i, y_j) in row j of the grid and on all of the black points $(x_i, y_{j\pm 1})$ in the preceding row $j\pm 1$ (the updates are paired). The \pm depends on the choice of direction. This fuses the red-black calculation so that we do a red-black ordered Gauss-Seidel with only one sweep across the grid instead of the standard two passes.

The update operations can be modified for SOR, SSOR, or ADI relaxation methods. Within the update operations we can further optimize the process by including Linpack style optimizations like loop unrolling to get 2 – 7 updates per iteration through the loop [2].

Some of the motivation behind this paper can be summarized in a simple example in two dimensions. Consider the grid in Figure 1, where the boundary points are

Standard	Cache aware
1. Do $j = 1, N$	1. $Update(1, \text{red})$.
1a. $Update(j, \text{red})$.	2. Do $j = 2, N$
2. Do $j = 1, N$	2a. $UpdateRedBlack(j)$.
2a. $Update(j, \text{black})$.	3. $Update(N, \text{black})$.

FIGURE 2. Standard and cache aware Gauss-Seidel

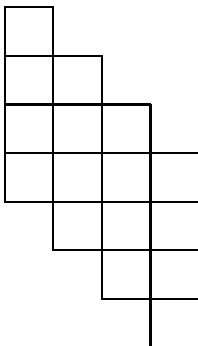


FIGURE 3. 5x5 rotated subdomain in 2D

included in the grid. Both standard and cache aware algorithms for the red-black ordered Gauss-Seidel iteration are given in Figure 2.

Each iteration of the standard algorithm, all of the data passes through the cache twice. Hence, with a small change in the algorithm’s implementation, the data only passes through cache once *per iteration*. However, the optimal number of times data should pass through cache is once for all iterations, which is a much harder algorithm to devise.

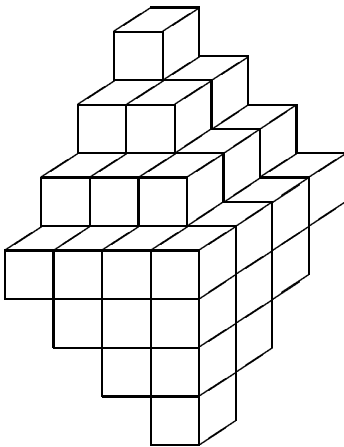
A more complicated approach is to use subdomains that move (i.e., an *active set* approach) across the domain. All computation is done inside of the subdomain. The subdomain is rotated and must be big enough so that once a grid vertex is no longer covered by the subdomain for the last time, the unknown(s) there are completely updated. Further, we require that the calculation be done in a particular way so that the solution using the fast approach is identical to what would have been calculated using a standard approach.

Requiring *bitwise compatibility* between the two solution approaches gives us a very useful debugging tool. Assume that $u^{(standard)}$ and u^{cache} are the computed approximations using the standard and the cache variants of the iterative algorithm, respectively. Then we know that

$$\|u^{(standard)} - u^{cache}\| = 0.$$

Any nonzero value of the norm indicates an error in the code. This also lets us know that the cache aware approach does not change the convergence rate of either the iterative method or the multigrid algorithm that we will use in §4.

Consider the 5x5 vertex rotated subdomain in two dimensions shown in Fig. 3. There is a small startup phase for the lower triangle of the grid, followed by a general update procedure inside of the subdomain only. The upper diagonal is updated, then the next lower diagonal, and so forth to the lowest diagonal. Then

FIGURE 4. $5 \times 5 \times 5$ rotated subdomain in 3D

the subdomain moves to the right until the rightmost part of the domain is updated. The subdomain moves back to the left, moved up one (natural ordering) or two rows (red-black ordering) and the process is repeated. The upper rows also require special handling.

The benefits of this procedure are two fold. First, only a fixed cache size is required, which can be precomputed easily. Second, the programming is actually easier than other approaches that have been suggested.

There is a drawback, however. In order to make this procedure run really fast, some assumptions and knowledge about the cache memory subsystem is required. If the data is allocated in large blocks of memory (ala Fortran), cache thrashing is a distinct possibility. Hence, padding arrays becomes necessary in order to avoid thrashing. Any scheme used in two dimensions can be defeated in three dimensions.

There are several approaches to solving this problem. Probably the best is to allocate the data differently than Fortran does. For example, use a list of allocated space instead of one large memory block. C and C++ naturally do this, resulting in inefficient memory usage and extra pointer chasing.

Using short memory blocks that are related to cache line sizes has been found to be particularly useful when using sparse matrices. This has not been explored in cache aware multigrid solvers to date, but will be as part of this group's current research project.

Now, how does this relate to solving three dimensional problems? The right approach in 3D is to take a rotated cube and move it through the domain as described before. Consider the $5 \times 5 \times 5$ vertex subdomain in three dimensions shown in Fig. 3. Once again a fixed cache size is needed. It is much bigger than the 2D case, however, and makes it much more difficult to fit into the small L1 caches offered by most vendors (HP being one of the few offering large L1 caches).

Table 1 contains the number of kilobytes needed in order to store a $N \times N \times N$ subdomain in cache at once. In fact, in order to compute efficiently and have some confidence that the cache algorithms actually keep data locked into cache, a much larger cache is required. Studies have shown that a program rarely gets the use of more than 60% of a cache memory (e.g., [13]).

Subdomain $N \times N \times N$	Constant coefficients	Variable coefficients	
	7 or 27 point	7 point	27 point
5	3	10	30
8	12	40	122
16	96	327	983

TABLE 1. Minimum number of kilobytes (KB) of L1 cache for 3D problems (twice the numbers ensures robustness)

Algorithm Vcycle($k, \{A_i, u_i, f_i, r_i\}_{i=1}^k$)

1. Do $i = 1, 2, \dots, k - 1$
 - 1a. Approximately solve $A_i u_i = f_i$.
 - 1b. Compute a residual $r_i \leftarrow f_i - A_i u_i$.
 - 1c. Set $f_{i+1} \leftarrow R_i r_i$ and $u_{i+1} \leftarrow 0$.
2. Do $i = k, k - 1, \dots, 1$
 - 2a. Approximately solve $A_i u_i = f_i$.
 - 2b. If $i > 1$, then set $u_{i-1} \leftarrow u_{i-1} + P_i u_i$.

FIGURE 5. V cycle definition

For 2D problems, a 16×16 subdomain appears to be optimal on most processors available. For 3D problems, this would require nearly 2 megabytes of L1 cache. Today, there is only one vendor/processor pair that meets this requirement (HP PA-8700).

4. Multigrid

A typical multigrid method is based on a V cycle multigrid method (see Figure 5). Implementing a W or F Cycle (or any other correction cycle) is a trivial extension.

Consider solving the following set of problems:

$$A_i u_i = f_i, \quad 1 \leq i \leq k,$$

where $u_i \in \mathbb{R}^{n_i}$ and $n_i > n_{i+1}$. Level 1 is the real problem that the solution is wanted on. All other levels are smaller, or coarser, approximations to level 1. The linear systems result from discretizing a partial differential equation over a given grid Ω_i . The discretization can be any standard finite element, difference, volume, or wavelet approach [1, 3, 5, 6]. Further, the discrete grids Ω_i will be assumed to be structured and regular (e.g., tensor product).

All multigrid correction algorithms are a simple combination of two distinct parts: the *pre-correction step* and the *post-correction step*. These are referred to by McCormick [11] as slash cycles. While both steps may have an approximate solve step included, the change of level step at the end of each has quite different cache effects.

There are 3 major operations in the V cycle:

- (1) Approximate solves: steps 1a and 2a. This is typically a relaxation method for simple problems, but can be any iterative method.
- (2) Restriction of residuals: steps 1b and 1c. This is typically a weighted average of nearby residuals and is represented by the operator R_i . It is

Algorithm Cache-Vcycle($k, \{A_i, u_i, f_i, r_i\}_{i=1}^k$)

1. Do $i = 1, 2, \dots, k - 1$
 - 1a. Do $\ell = 1, 2, \dots, m_i$
 - 1a1. Determine $\Omega_{ij}^{(\ell)}$.
 - 1a2. For each $\Omega_{ij}^{(\ell)}$,
 - 1a2a. If $\ell = 1$, then $u_i|_{\Omega_{ij}^{(\ell)}} \leftarrow 0$.
 - 1a2b. Do 1 iteration of approximate solve of $A_i u_i = f_i$.
 - 1a2c. If $\ell = m_i$, then compute as much of r_i as is possible.
 - 1a3. If $\ell = m_i$, then complete the calculation of r_i and calculate $f_{i+1} = R_i r_i$.
2. Do $i = k, k - 1, \dots, 1$
 - 2a. Do $\ell = 1, 2, \dots, m_i$
 - 2a1. Determine $\Omega_{ij}^{(\ell)}$.
 - 2a2. For each $\Omega_{ij}^{(\ell)}$,
 - 2a2a. If $\ell = 1$ and $k = 1$, then $u_i|_{\Omega_{ij}^{(\ell)}} \leftarrow 0$.
 - 2a2a. If $\ell = 1$ and $k > 1$, then $u_i|_{\Omega_{ij}^{(\ell)}} \leftarrow u_i|_{\Omega_{ij}^{(\ell)}} + P_{i+1} u_{i+1}|_{\Omega_{ij}^{(\ell)}}$.
 - 2a2c. Do 1 iteration of approximate solve of $A_i u_i = f_i$.

FIGURE 6. Cache aware V cycle definition

used to compute a residual correction problem's right hand side on the next coarser grid.

- (3) Prolongation of corrections: step 2b. This is typically a second or fourth order interpolation method (represented by the operator P_i).

In a typical multigrid code, a V cycle is implemented very similarly to the description given here. By using a structured language methodology, different algorithms can be substituted for the default ones trivially.

As was shown in [4], when a grid Ω_i gets too large, a change in the algorithm is required which changes the global ordering. In essence, we use a domain decomposition approach to find disjoint two dimensional subdomains Ω_{ij} whose union is Ω_i . Further, the data associated with the Gauss-Seidel operation on each subdomain must fit entirely in cache. The size of the subdomains depends heavily on the number of nonzeros per row in the matrix A_i , the sparse matrix storage method, and what iteration of the Gauss-Seidel iteration we are on. Hence, we really have a set of disjoint subdomains $\Omega_{ij}^{(\ell)}$, where $\ell = 1, \dots, m$.

Data passes through cache once almost everywhere each time a level is reached with this transformation. Due to connections between subdomains, sometimes a very few points (rather than whole subdomains) have data pass through cache twice. However, we can do better, as will be described in this and the next section.

The computational subdomains $\Omega_{ij}^{(\ell)}$ must be further refined in order to use the largest subdomains possible for each iteration of the relaxation algorithm. The last iteration of the relaxation algorithm must be treated differently due to the projection or interpolation steps that must be done.

Determining the sizes of the $\Omega_{ij}^{(\ell)}$'s per iteration ℓ can be done as a preprocessing step and is inexpensive. In order to efficiently do loop unrolling and/or tiling, we

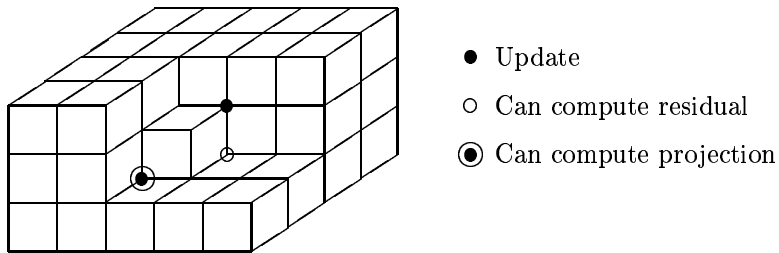


FIGURE 7. Seven point discretization, point relaxation

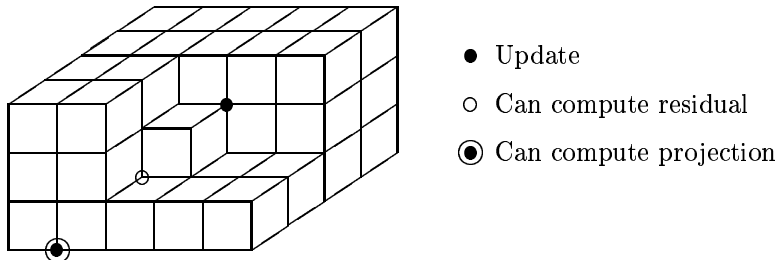


FIGURE 8. Twenty seven point discretization, point relaxation

need a small amount of information about the computer that we are going to use. First, we need to know how big the usable part of the cache actually is. As is noted in [13], only 50-60% of the cache is actually available for use by a given program. This is a side effect of multitasking operating systems. Knowing the size of usable cache and the number of points in a line we calculate

- $\Omega_{ij}^{(1)}$ for either a precorrection or a postcorrection step.
- $\Omega_{ij}^{(\ell)}$ for $\ell = 2, \dots, m_i - 1$.
- $\Omega_{ij}^{(m_i)}$ for either a precorrection or a postcorrection step.

The second item we need to know is how many points to unroll in the loops in the *Update* and *UpdateRedBlack* code.

Three steps occur during the pre-correction step: smoothing, residual calculation, and projection. Two sets of computational subdomains $\Omega_{ij}^{(\ell)}$ are necessary. One is for when just the relaxation method is used as a smoother and the other is when all three components are used at once.

There is a scheduling issue when implementing cache aware multigrid algorithms. We can graphically show when the residual can be computed based on the last update in the relaxation method. Where a projection can be centered is also shown based on which residuals have been computed.

Scheduling for a seven point operator is given in Figs. 7 and 8. Scheduling for interpolation to the next finer level is given in Fig. 9. What is expressed is simply a “compute as soon as you can” principal.

In order to write highly efficient code with no special cases, two extra “ghost” elements are needed to surround the computational grids. Values there are set to zero and no relaxation updates occur (requiring a minor post processing). While for

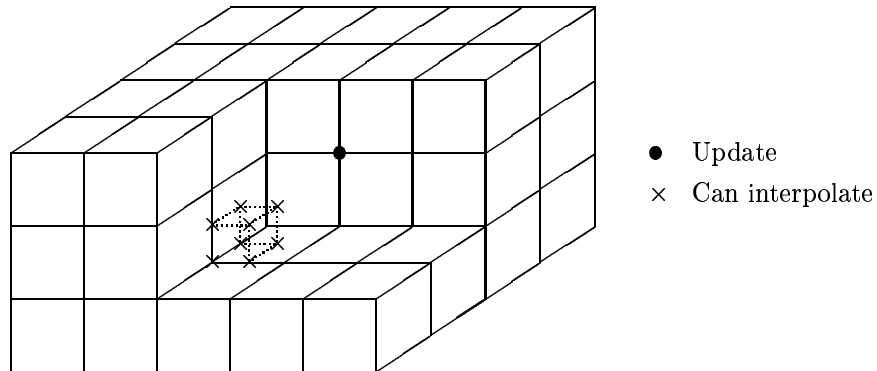


FIGURE 9. Twenty seven point discretization, point relaxation

tiny grids this adds a significant amount of storage, we are only interested in problems that are much larger than would fit in the cache. Hence, the padding on the coarser grids only amounts to a trivial increase in the overall storage requirements (and is useful for parallel computing versions of the algorithms).

Two steps occur in the post-correction step: smoothing and interpolation. In order for this half of a correction scheme to be optimally cache aware, the last step must use a somewhat smaller $\Omega_{ij}^{(m_i)}$ than the rest of the iterations. This is because the interpolant on level $i + 1$ is added to a much larger vector, which is typically four times the size of the vector on level i .

5. Performance and Conclusions

Using the algorithms in this paper, the performance can be dramatically improved. Assume that the problems are large enough so that the data does not all fit into cache. A typical multigrid implementation runs at 2 – 5% of the peak speed of a processor. A typical moving subdomain approach runs at about 20 – 30% of peak speed [14].

Reducing the number of times data passes through cache to the absolute minimum eliminates one of the major areas where multigrid does not take full advantage of the hardware that it is implemented on. The algorithms in this paper reach this minimum for problems on tensor product grids in three dimensions.

When using cache aware implementations, there are two issues that must be faced: is true portability wanted and how much performance is demanded? By true portability, all that must be determined for a given machine are two numbers: the number of elements in a cache line and the number of cache lines that can be guaranteed to be usable.

If ultra high performance is demanded, true portability becomes much harder. Loop unrolling, relaxation updates along diagonals, specialty BLAS for short vectors (in machine language), and other machine specific optimizations are necessary. One interesting aspect is that most RISC based processors are very similar. Hence, the non-machine language optimizations will work well on similar processors.

Using cache aware multigrid algorithms requires a different programming style than is traditional. In order to make the codes viable, a rigid programming style

must be maintained for all components of the code. This includes a uniform subscripting method for variables, an isolation of the minimal number of lines of code that is necessary for a given discretized problem to do major components (e.g., updating a point by the relaxation method), and a decision on how portable the code needs to be.

If only true portability is demanded, a general code can be constructed quite easily for 7 or 27 point operators A_i . For the examples in §4, only four quite small pieces of code have to be written as separate “include” files or macro definitions. Only the interpolation file will typically have more than one target point to modify.

The ideas in this paper can be applied not just to natural and red-black ordered Gauss-Seidel, but to SOR variants, line relaxation methods, and ADI. Applying the ideas to typical parallel smoothers is also a straight forward extension and is being actively worked on.

References

1. R. E. Bank and C. C. Douglas, *Sharp estimates for multigrid rates of convergence with general smoothing and acceleration*, SIAM J. Numer. Anal. **22** (1985), 617–633.
2. J. Dongarra and R. C. Whaley, *Automatically tuned linear algebra software*, In URL <http://www.netlib.org/atlas>, 1999.
3. C. C. Douglas, *Multi-grid algorithms with applications to elliptic boundary-value problems*, SIAM J. Numer. Anal. **21** (1984), 236–254.
4. ———, *Caching in with multigrid algorithms: problems in two dimensions*, Paral. Alg. Appl. **9** (1996), 195–204.
5. C. C. Douglas and J. Douglas, *A unified convergence theory for abstract multigrid or multilevel algorithms, serial and parallel*, SIAM J. Numer. Anal. **30** (1993), 136–158.
6. C. C. Douglas, J. Douglas, and D. E. Fyfe, *A multigrid unified theory for non-nested grids and/or quadrature*, E. W. J. Numer. Math. **2** (1994), 285–294.
7. C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rude, and C. Weiss, *Portable memory hierarchy techniques for pde solvers, part i*, SIAM News **33** (2000), no. 5, 1, 8–9.
8. ———, *Portable memory hierarchy techniques for pde solvers, part ii*, SIAM News **33** (2000), no. 6, 1, 10–11, 16.
9. C. C. Douglas, J. Hu, M. Kowarschik, U. Rude, and C. Weiss, *Cache optimization for structured and unstructured grid multigrid*, Elect. Trans. Numer. Anal. **10** (2000), 21–40.
10. J. Handy, *The cache memory book*, Academic Press, New York, 1998.
11. S. F. McCormick, *Multigrid methods for variational problems: general theory for the V-cycle*, SIAM J. Numer. Anal. **22** (1985), 634–643.
12. D. A. Patterson and J. L. Hennessy, *Computer architecture: A quantitative approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.
13. J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, *Thread scheduling for cache locality*, Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems (Cambridge, MA), ACM, 1996, pp. 60–73.
14. C. C. Douglas U. Rude, , J. Hu, M. Kowarschik, W. Karl, H. Pfaender, L. Stals, D. T. Thorne, and C. Weiss, *Cache based iterative methods*, SC2001, IEEE, Los Alamitos, 2001.

UNIVERSITY OF KENTUCKY, DEPARTMENT OF COMPUTER SCIENCE, 325 McVEY HALL-CCS, LEXINGTON, KY 40506-0045, USA AND YALE UNIVERSITY, DEPARTMENT OF COMPUTER SCIENCE, P.O. BOX 208285, NEW HAVEN, CT 06520-8285, USA.

E-mail address: `douglas@ccs.uky.edu` or `douglas-craig@cs.yale.edu`

UNIVERSITY OF KENTUCKY, DEPARTMENT OF COMPUTER SCIENCE, 325 McVEY HALL-CCS, LEXINGTON, KY 40506-0045, USA

E-mail address: `thorne@ccs.uky.edu`