Maximizing Cache Memory Usage for Multigrid Algorithms

Craig C. Douglas Jonathan Hu
Mohamed Iskandarani Markus Kowarschik
Ulrich Rüde Christian Weiss

Abstract

Computers today rely heavily on good utilization of their cache memory subsystems. Compilers are optimized for business applications, not scientific computing ones, however. Automatic tiling of complex numerical algorithms for solving partial differential equations is simply not provided by compilers. Thus, absolutely terrible cache performance is a common result.

Multigrid algorithms combine several numerical algorithms into a more complicated algorithm. In this paper, an algorithm 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.

KEYWORDS: multigrid, cache, threads, sparse matrix, iterative methods, domain decomposition, compiler optimization.

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 speed ups in the hardware. Essentially all

Lecture Notes in Physics
Chen, Ewing, and Shi (eds.), pp. 1–15.

put here ISBN

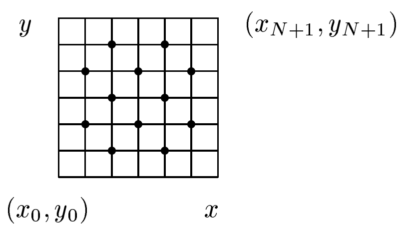


Figure 1: Simple grid with red points marked

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. 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 lines may hold data from quite separate parts of main memory.

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.

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 [11].

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 grid:

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

and

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

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

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

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 or reverse. The natural order is assumed unless a direction is given. 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 $(x_i, y_{j\pm1})$ in the preceding row $j\pm1$ (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.

Some of the motivation behind this paper can be summarized in a simple example. Consider the grid in Figure 1, where the boundary points are 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. Unfortunately, no compiler for commonly used languages (e.g., Fortran, C, or C++) seems to exist that can optimize the first form of the red-black ordered Gauss-Seidel algorithm automatically into the second form.

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 speed up for a modest amount of work.

2 Relaxation Methodology and Notation

Consider solving the following set of problems:

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

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 (see [1] and [2] for examples of this approach). Further, the discrete grids Ω_i will be assumed to be structured and regular.

2.1 Once through cache naturally ordered Gauss-Seidel

Consider the naturally ordered Gauss-Seidel first. Let us restrict our attention to matrices A_j which are based on discretization methods which are local to only 3 neighboring rows of the grid (e.g., a 5 or 9 point discretization). We have to assume that $\ell + m - 1$ rows of a $N \times N$ grid G fit entirely into cache simultaneously and that $m < \ell$.

Figure 3 contains the complete algorithm for passing data through cache only once for the naturally ordered Gauss-Seidel. There are two special cases to the cache aware algorithm: the first block of rows and the rest of the blocks.

The first case is for the first ℓ rows of the grid. At the end of step 1 in Figure 3, the data associated with rows 1 to ℓ has been brought into cache

Algorithm Cache-GSNat

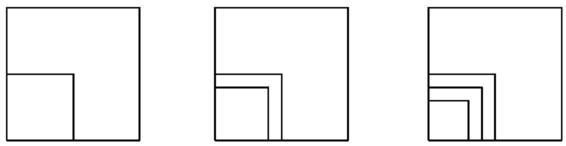
1. Do $it = 0, m-1$
   - 1a. Do $i = 1, \ell - it$
     - 1a1. $Update(i, \text{all})$.
2. Do $j = \ell + 1, N, \ell$.
   - 2a. Do $it = 0, m-1$
     - 2a1. Set $j_1 = \min(j+\ell, N)$.
     - 2a2. If $j + \ell < N$, then
       - 2a2a. $j_2 = \max(j_1 - it - 1, j)$.
     - 2a3. Else
       - 2a3a. $j_2 = N$
     - 2a4. Do $i = j, j_2$
       - 2a4a. $Update(i, \text{all})$.
     - 2a5. Do $i = j-1, j+it-m+1, -1$
       - 2a5a. $Update(i, \text{all})$.

Figure 3: Once through cache naturally ordered Gauss-Seidel algorithm

only once, not m times. The data in rows 1 to $\ell - m + 1$ has been updated m times. The data in rows j, $\ell - m + 2 \leq j \leq \ell$ has been updated $\ell - j + 1$ times.

Once the first block of grid rows is partially updated, we have a new block to update and must also finish updating the previous block of grid rows. This corresponds to step 2 in Figure 3. Note that the second inner loop (step 2a3) runs in the opposite order as the first inner loop (step 2a2), which ensures that the updates are bitwise identical to the standard algorithm. In effect, we have applied a domain decomposition methodology to the standard iteration. Each iteration requires a new, slightly smaller domain.

The cache aware algorithm can be generalized to grids which are too large to fit entire rows into cache. In this case, a multidimensional approach is necessary when dropping rows and columns from the subdomains. In each of the three pictures below, computation occurs in the smallest subdomain only (the outer parts represent the buffers that have been added). The buffers grow by one line in each of x and y each iteration.



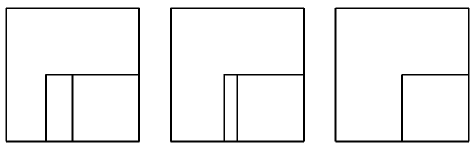
First iteration　Second iteration　Third iteration

Algorithm Cache-GSRedBlack

1. Do $j = 1, N, \ell$.
    - 1a. Set $j_1 = \min(j + \ell - 1, N)$.
    - 1b. Do $it = 0, m - 1$
        - 1b1. Do $i = j, j_1 - 2 * it$
            - 1b1a. $Update(i, \text{red})$.
            - 1b1b. $Update(i - 1, \text{black})$.
        - 1b2. Do $i = j - 1, j - m + 2 * it, -2$
            - 1b2a. If mod(j,2)= 1 then
                - 1b2a1. $UpdateRedBlack(i)$
                - 1b2a2. $UpdateRedBlack(i - 1)$
            - 1b2b. If mod(j,2)= 0 then
                - 1b2b1. $UpdateRedBlack(i - 1)$
                - 1b2b2. $UpdateRedBlack(i)$

Figure 4: Once through cache red-black ordered Gauss-Seidel algorithm

While computing in neighboring subdomains, the rest of the updates are done carefully in order to maintain bitwise identical updates. In the current subdomain, the calculation is done using the natural ordering. In the trailing subdomain, the calculation is done in the opposite order. In the three pictures below, the shrinking subdomain on the left is the trailing subdomain.



2.2 Once through cache red-black ordered Gauss-Seidel

Red-black ordered Gauss-Seidel is significantly more complex to code in the once through cache style rather than in the normal style. Let us restrict our attention to matrices A_j which are based on discretization methods which are local to only 3 neighboring rows of the grid (e.g., a 5 or a 9 point discretization). For a discussion of a 9 point discretization on this topic, see [9] and [12].

Figure 4 contains the complete algorithm for passing data through cache only once for the red-black ordered Gauss-Seidel. In §2.1, the naturally ordered Gauss-Seidel was shown geometrically to be similar to a domain de-

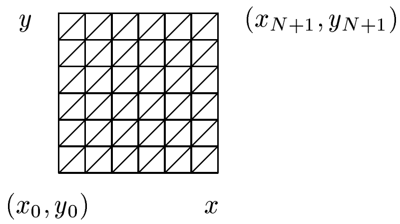


Figure 5: Simple triangular grid

composition method where the size of the subdomains shrank each iteration (forming an increasingly larger buffer each iterations). For the red-black ordered Gauss-Seidel, the buffer is saw tooth shaped on the top and grows each iteration by points from two grid rows instead of one. On the right side, the buffer grows each iteration by only one grid column (similar to the naturally ordered Gauss-Seidel case).

2.3 Once through cache ADI and line relaxation methods

ADI and line relaxation algorithms are easily made cache aware using similar techniques to the ones given in §§2.1-2.2. One difference is that entire lines of unknowns must fit in cache at once in order to guarantee bitwise compatibility with the standard algorithm implementations.

2.4 Once through cache relaxation methods on triangular grids

Triangular grids are also easily made cache aware. The grids must be quasi-uniform and highly structured and the ordering of A_i must lead to blocks of diagonal submatrices. A simple example of a suitable grid is given in Figure 5. A combination of the methods given in §§2.1-2.2 with the nonzero graph of the A_i's is used.

Grids that are unstructured, including ones that have been chosen through an adaptive gridding procedure, require a different approach than is covered in this paper. §4 briefly discusses an unstructured grid approach.

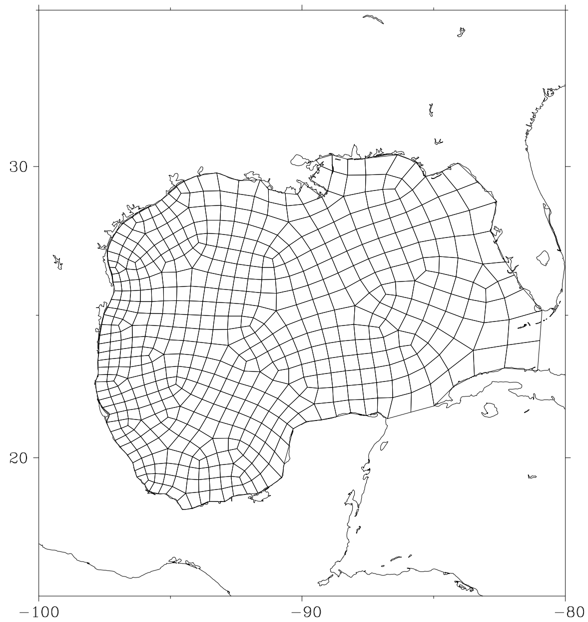


Figure 6: Quasi-structured grid.

2.5 Unstructured and Quasi-Structured Grids

Unstructured grids presents a challenge that is not addressed in this paper. Quasi-unstructured grids (see Figure 6) can frequently be accommodated using techniques similar to structured grids. In particular, the number of graph connections in the matrices A_i are usually predictable, just like in the structured grid case. Both of these cases are considered in [5], [6], and [7].

3 Combining Multigrid Components

A typical multigrid method is based on a V cycle multigrid method (see Figure 7). Implementing a W or F Cycle (or any other correction cycle) is a trivial extension.

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 [10] 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:

Algorithm Vcycle(k, $\{A_i, u_i, f_i, r_i\}_{i=1}^{k}$)
1. Do $i = 1, 2, \cdots, k-1$
   - 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 $i = k, k-1, \cdots, 1$
   - 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 7: V cycle definition

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. It is used to compute a residual correction problem's right hand side (represented by the operator R_i) 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 [3], 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 nonzeroes 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 $\ell = 1, \cdots, m$.

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.

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

1. Do $i = 1, 2, \cdots, k-1$
   - 1a. Do $\ell = 1, 2, \cdots, m_i$
     - 1a1. Determine $\Omega_{ij}^{(\ell)}$.
     - 1a2. For each $\Omega_{ij}^{(\ell)}$,
       - 1a2a. If $\ell = 1$, then $u_i \mid_{\Omega_{ij}^{(\ell)}} \leftarrow 0$.
       - 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 $i = k, k-1, \cdots, 1$
   - 2a. Do $\ell = 1, 2, \cdots, m_i$
     - 2a1. Determine $\Omega_{ij}^{(\ell)}$.
     - 2a2. For each $\Omega_{ij}^{(\ell)}$,
       - 2a2a. If $\ell = 1$ and $k = 1$, then $u_i \mid_{\Omega_{ij}^{(\ell)}} \leftarrow 0$.
       - 2a2a. If $\ell = 1$ and $k > 1$, then $u_i \mid_{\Omega_{ij}^{(\ell)}} \leftarrow u_i \mid_{\Omega_{ij}^{(\ell)}} + P_{i+1} u_{i+1} \mid_{\Omega_{ij}^{(\ell)}}$.
       - 2a2c. Do 1 iteration of approximate solve of $A_i u_i = f_i$.

Figure 8: Cache aware V cycle definition

The computational subdomains $\Omega_{ij}^{(\ell)}$ must be further refined in order to use as large of subdomains as possible 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. As is noted in [11], only 50-60% of the cache is actually available for use by a given program. This is a side effect of multitasking operating systems.

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

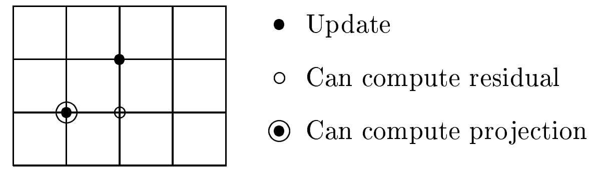


Figure 9: Five point discretization, point relaxation

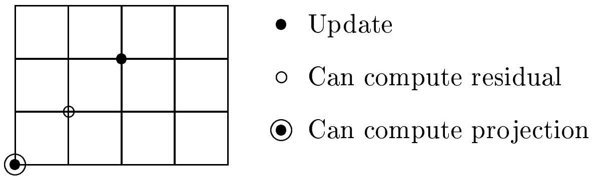


Figure 10: Nine point discretization, point relaxation

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. Figures 9-11 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. Figure 12 shows when interpolation to the next finer level can be scheduled. What is expressed is simply a “compute as soon as you can” principal.

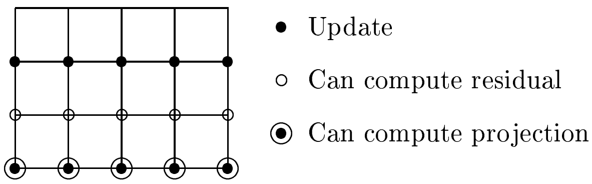


Figure 11: Five or nine point discretization, line relaxation

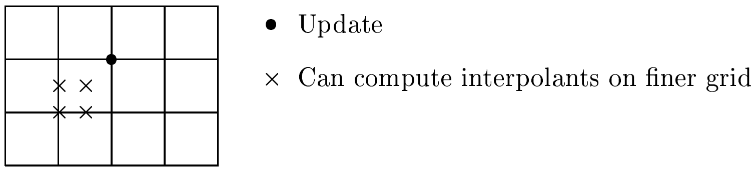


Figure 12: Five or nine point discretization, interpolation

For simplicity, point relaxation methods should probably be written assuming a nine point operator (see Figure 11). In order to write highly efficient code with no special cases, two extra “ghost” rows and columns are needed in the computational grids. Values there are set to zero and no relaxation updates occur (requiring a minor post processing). While for 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 much larger vector, which is typically four times the size of the vector on level i.

4 Numerical Results and Conclusions

In [5], [6], and [7] a collection of problems are solved on structured grids (2D and 3D) and unstructured grids (in 2D). Speedups range from 100% to 300% over using standard, well coded implementations.

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 show how to reach this minimum.

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 (see [4] and [8]). 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 give discretized problem to do major components (e.g., updating a point by the relaxation method), and a decision on how portable the code will be.

If only true portability is demanded, a general code can be constructed quite easily for 5-9 point operators A_i (see [4] for example codes). For the examples in §3, 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.

Acknowledgments. This research was supported in part by the Deutsche Forschungsgemeinschaft (project Ru 422/7-1), the National Science Foundation (grants DMS-9707040 and ACR-9721388), NATO (grant CRG 971574), and the National Computational Science Alliance (grant OCE980001N and utilized the NCSA SGI/Cray Origin2000).

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.*, 22:617–633, 1985.

[2] C. C. Douglas. Multi–grid algorithms with applications to elliptic boundary–value problems. *SIAM J. Numer. Anal.*, 21:236–254, 1984.

[3] C. C. Douglas. Caching in with multigrid algorithms: problems in two dimensions. *Paral. Alg. Appl.*, 9:195–204, 1996.

[4] C. C. Douglas. Reusable cache memory object oriented multigrid algorithms. See http://www.ccs.uky.edu/~douglas under *preprints*, 1999.

[5] C. C. Douglas, J. Hu, and M. Iskandarani. Preprocessing costs of cache based multigrid. In *Proceeding of ENUMATH99: Third European Conference on Numerical Methods for Advanced Applications*, 8 pages, Singapore, 2000. World Scientific.

[6] C. C. Douglas, J. Hu, W. Karl, M. Kowarschik, U. Rüde, and C. Weiss. Fixed and adaptive cache aware algorithms for multigrid methods. In *European Multigrid VI*, Lecture Notes in Computational Science and Engineering, 7 pages. Springer, Berlin, 2000.

[7] 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.

[8] C. C. Douglas, J. Hu, U. Rüde, and M. Bittencourt. Cache based multigrid on unstructured two dimensional grids. Notes on Numerical Fluid Mechanics, 11 pages. Vieweg, Braunschweig, 1999. Proceeding of the 14th GAMM-Seminar Kiel on 'Concepts of Numerical Software' (January, 1998).

[9] H. Hellwagner, C. Weiß, L. Stals, and U. Rüde. Efficient implementation of multigrid on cache based architectures. Notes on Numerical Fluid Mechanics. Vieweg, Braunschweig, 1999. Proceeding of the 14th GAMM-Seminar Kiel on 'Concepts of Numerical Software' (January, 1998).

[10] S. F. McCormick. Multigrid methods for variational problems: general theory for the V–cycle. *SIAM J. Numer. Anal.*, 22:634–643, 1985.

[11] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60–73, Cambridge, MA, 1996. ACM.

[12] L. Stals and U. Rüde. Techniques for improving the data locality of iterative methods. Technical Report MRR 038–97, Australian National University, 1997.

Craig C. Douglas
Department of Computer Science
University of Kentucky
325 McVey Hall - CCS
Lexington, KY 40506-0045, USA
douglas@ccs.uky.edu

Jonathan Hu
Department of Mathematics
University of Kentucky
715 Patterson Office Tower
Lexington, KY 40506-0027, USA
jhu@ms.uky.edu

Mohamed Iskandarani
Institute of Marine and Coastal Sciences
Rutgers University
P.O. Box 231
New Brunswick, NJ 08903-0231, USA
mohamed@ahab.rutgers.edu

Markus Kowarschik
Lehrstuhl für Systemsimulation (IMMD 10)
Institut für Informatik
Universität Erlangen-Nürnberg
Martensstrasse 3, D-91058 Erlangen, Germany
kowarschik@informatik.uni-erlangen.de

Ulrich Rüde
Lehrstuhl für Systemsimulation (IMMD 10)
Institut für Informatik
Universität Erlangen-Nürnberg
Martensstrasse 3, D-91058 Erlangen, Germany
ruede@informatik.uni-erlangen.de

Christian Weiss
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik
Technische Universität München
D-80290 München, Germany
weissc@in.tum.de