On Optimal Temporal Locality of Stencil Codes

Claudia Leopold
Friedrich-Schiller-Universität Jena
Institut für Informatik
07740 Jena, Germany
claudia@informatik.uni-jena.de

Keywords
Data locality, tiling, relaxation methods, lower bounds

ABSTRACT
Iterative solvers such as the Jacobi and Gauss-Seidel relaxation methods are important, but time-consuming building blocks of many scientific and engineering applications. The performance problems are largely due to cache misses, and can be reduced by tiling the codes. Whereas previous research has shown the usefulness of tiling by experimentally comparing the run times of tiled and original codes, it did not tackle the question as to whether further improvements are possible. In this paper, we give a negative answer, regarding the exploitation of temporal locality in one step of a 2-dimensional stencil code. We derive upper and lower bounds that match up to a factor of about $1 + 2/M$, where M is the cache size. For the upper bounds, we investigate some modifications of tiling.

1. INTRODUCTION
Many scientific and engineering applications require the solution of partial differential equations (PDEs). A common approach discretizes the input domain, thereby transforming the PDE into a sparse system of linear equations, and then solves the system iteratively. Classical iterative solvers include the Jacobi and Gauss-Seidel relaxation methods, which have nowadays been largely replaced by more efficient schemes such as multigrid. Nevertheless, the classical methods remain important, because they are building blocks of the advanced schemes, and because they have similar computational properties.

The Jacobi and Gauss-Seidel methods are frequently denoted as *stencil codes*, because they update array elements according to some fixed pattern, called stencil. The kernels are depicted in Fig. 1. As the figure shows, stencil codes perform a sequence of sweeps over a large array; in each sweep, all array elements are updated using the values of a few neighbors. In Fig. 1, for instance, it is four neighbors plus the array element itself so that we speak of a five-point stencil.

SAC 2002, Madrid, Spain

As Fig. 1 shows, the Jacobi scheme updates the array elements based on neighbors from the last iteration, whereas the Gauss-Seidel scheme uses neighbors from both the present and last iterations. Consequently, within a sweep, there are no data dependencies between updates (*) in the case of Jacobi, but there are data dependencies in the case of Gauss-Seidel. In other words, the Gauss-Seidel scheme imposes more constraints on the execution order, and thus exhibits less opportunities for optimization through reordering.

Stencil codes are among the most time-consuming routines in many applications, and thus it makes sense to strive for ultimate performance. As has been observed by, for instance, Douglas et al. [1], it is, in particular, cache misses that slow down the execution of stencil codes. How stringent the problem is depends on cache size, array size, and dimensionality of the array [6].

Even without optimization, stencil codes exhibit a certain degree of locality. Consider the Gauss-Seidel code in Fig. 1a), and assume column-major storage order. First there is *spatial locality* because the accesses to $A(I-1,J)$, $A(I,J)$, $A(I+1,J)\ldots$ in the inner loop refer to the same cache line (except at line boundaries).

Second, $A(I,J)$ is used in the updates of $A(I-1,J)$, $A(I,J)$, and $A(I+1,J)$. Since the I-loop is innermost, $A(I,J)$ remains in cache in-between these three updates. This is denoted as *temporal locality* because it is the *same* array element that is reused (as opposed to different elements from the same cache line in the case of spatial locality).

Third, if the cache is large enough to hold two columns of A (plus a few elements), $A(I,J)$ also remains in cache in-between the updates of $A(I,J-1)$, $A(I,J)$, and $A(I,J+1)$. This use of temporal locality in the second dimension is often exploited by L2 caches, but less often by L1 caches. Tiling is a well-known program transformation that exploits this type of reuse for small caches, too, as we will see in Sect. 4.

Throughout the paper, we restrict ourselves to the investigation of temporal locality, by assuming line size one. Furthermore we ignore the possibility of thrashing, that is the fact that limited cache associativity may lead to conflict misses. Thrashing can be avoided through tile size selection and padding, as it is discussed, for instance, in [6]. These techniques complement locality optimization.

The paper supposes a cache size less than N, so that temporal locality in the second dimension is not exploited by the input codes. We derive upper and lower bounds for the number of cache misses. The lower bounds are new, and the upper bounds are based on tiling [9,

```
a)  do  T = 1,  time
     do  J = 1,  N-1
      do  I = 1,  N-1
       A(I,J)   = W1*A(I-1,J)    + W2*A(I+1,J)
              + W3*A(I,J-1)    + W4*A(I,J+1)
              + W5*A(I,J);                  (*)
      end  do
     end  do
    end  do

b)  do  T = 1,  time
     do  J = 1,  N-1
      do  I = 1,  N-1
       B(I,J)   = W1*A(I-1,J)    + W2*A(I+1,J)
              + W3*A(I,J-1)    + W4*A(I,J+1)
              + W5*A(I,J);                  (*)
      end  do
     end  do
     do  J = 1,  N-1
      do  I = 1,  N-1
       A(I,J)   = W1*B(I-1,J)    + W2*B(I+1,J)
              + W3*B(I,J-1)    + W4*B(I,J+1)
              + W5*B(I,J);                  (*)
        end  do
      end  do
    end  do
```

Figure 1: a) Gauss-Seidel, and b) Jacobi code. A **and** B **are assumed to be** $[0 \dots N] \times [0 \dots N]$ **arrays.**

6]. The bounds match up to a factor of $1 + \epsilon/M$ for cache size M, where ϵ differs between 2.1 and 4.2 depending on the particular code. The upper bounds incorporate some minor modifications of tiling, which slightly reduce the number of cache misses, but do not lead to overall performance gains.

Briefly stated, the paper shows that tiling works well for stencil codes, and that it is not worthwhile to investigate locality optimization any further. The result refers to one sweep through the array and assumes that (*) is considered a unit. The latter assumption has been made to guarantee bitwise-identical results as compared to the original codes.

Section 2 of the paper presents the lower bound proof, which applies to both the Gauss-Seidel and the Jacobi schemes. We first derive an asymptotic bound, and afterwards estimate the constant factor. Section 3 deals with the upper bound. We present and analyze the tiled algorithms, and discuss modifications that further improve the constant. Section 4 surveys related work, and Section 5 finishes with conclusions.

2. LOWER BOUND

We consider one sweep through the array, that is, one iteration of the outer time loop for Gauss-Seidel, or one execution of the, say, upper loop nest for Jacobi, and take each update (*) as a unit. Furthermore, we assume cache size M and line size 1.

Our argumentation resembles the red-blue pebble game of Hong and Kung [2] insofar as that we argue on the best schedule of a program dag. The program dag is the directed acyclic graph whose nodes are marked with the updates (*) for $I, J = 1 \dots N-1$, and whose arcs reflect data dependencies. Data dependencies exist from iterations $(I{-}1, J)$ to iterations (I, J), and from iterations $(I, J{-}1)$ to iterations (I, J) in the Gauss-Seidel scheme. The lower bound proof, actually, does not take the data dependencies into account, which is correct because a lower bound to the unconstrained problem is always a lower bound to the constrained problem, too.

Similarly, we do not rely on a particular cache replacement scheme. The lower bound refers to user-controlled data placement, but, consequently, also holds for any common scheme such as LRU.

In the literature, cache misses are frequently classified as cold misses (the first access to an element), capacity misses (misses due to limited cache capacity), and conflict misses (misses due to thrashing). Independent of optimization, the Gauss-Seidel and Jacobi schemes always take $(N+1)^2 - 4$ cold misses since all elements of A (except the corners) must be read. Here we analyze the number of capacity misses. We speak of redundant computations if, for any I, J, update (*) is executed more than once.

THEOREM 2.1. *Any schedule of the dag, including schedules with redundant computations, takes* $\Omega(N^2/M)$ *capacity misses.*

PROOF. Let $Sched$ be any schedule, that is, any assignment of dag nodes to execution times. Since the operations (*) have equal lengths, $Sched$ can also be considered as an operation sequence. We partition *Sched* into subsequences $S_1, S_2, \dots S_k$ of successive operations such that $S_1, S_2, \dots S_{k-1}$ consist of exactly M^2 operations, and S_k consists of up to $2M^2$ operations. Note that $k \geq \lfloor (N-1)^2/M^2 \rfloor$, where inequality holds for redundant computations.

The codes under consideration typically take 5 accesses per array element. If the element belongs to the boundary of A, it is less, if redundancy is used, it is more accesses. We say that an element e that is accessed by S_i is *touched only* by S_i if

- the number of accesses to e in S_i is less than the total number of accesses to e in $Sched$, or
- e belongs to the boundary of A, or
- a neighbor of e belongs to the boundary of A.

LEMMA 2.2. *By any* S_i*, at least* $3M$ *array elements are touched only.*

PROOF. (Lemma 2.2): The proof will use the fact that the elements accessed in (*) are neighbors, and it will deploy a geometric argumentation. Therefore, we model the array by a (square) rectangle that is composed of small squares for the $A(I, J)$. Just think of the rectangle as being drawn on squared paper such that each $A(I, J)$ corresponds to the printed square in the I-th row and J-th column of the rectangle.

Now consider any particular S_i, and let F denote the geometric figure that corresponds to S_i, that is, the figure that consists of those printed squares whose array elements are updated in S_i. Furthermore, let R be the smallest axes-parallel rectangle that completely holds F. Obviously, the length l and width w of R fulfill

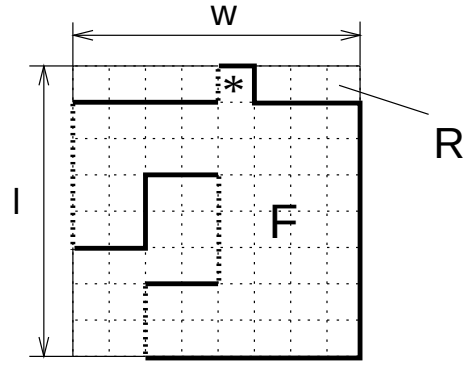

Figure 2: Minimum boundary length of F. For illustration, the boundary segments on the left are printed as dashed lines.

$l \cdot w \geq M^2$. Here, we take the side length of a printed square as the unit of length, and the printed square itself as the unit of area.

Next we show that the boundary of F has a certain minimum length. Without loss of generality, we assume that F is connected; otherwise closing up the parts of F would shorten the boundary. By *segment*, we denote a side of a printed square. Since R is minimum, F has at least l vertical boundary segments on the left, l vertical boundary segments on the right, w horizontal boundary segments on the top, and w horizontal boundary segments on the bottom (see Fig. 2). This makes up for a total boundary length of at least $2l + 2w \geq 2M^2/w + 2w = f(w)$.

The minimum of function f is easily determined by elementary analysis. Since $f'(w) = -2M^2/w^2 + 2$, there is only one local optimum within the interval $1 \leq w \leq M^2$, namely $w = M$. The corresponding value is $f(M) = 4M$, which is less than $f(1)$ and $f(M^2)$. Consequently, the boundary of F has length at least $4M$.

Going back from the geometric model to the codes, an update of $A(I, J)$ at the, say, left boundary of S_i involves $A(I-1, J)$, but $A(I-1, J)$ is touched only since the update of $A(I-1, J)$ does not belong to S_i. This argumentation does not apply to the $A(I, J)$ at the left boundary of A, but these data are trivially touched-only by definition. Hence, at any boundary segment, two array elements are touched only: one outside S_i (here $A(I-1, J)$), and one inside S_i (here $A(I, J)$). The number of touched-only elements is less than twice the total boundary length, however, since squares adjoin up to four segments. For instance, the square marked '*' in Fig. 2 adjoins the segments to the left, right, and above it. The claimed value of $3M$ follows from the observation that $l \geq M$ or $w \geq M$. Assuming $l \geq M$, in each row at least 3 elements are touched by vertical segments. Consequently, at least $3M$ elements are touched-only by S_i. □

PROOF. (Theorem 2.1, continued): The number of pairs (S_i, e) with S_i touching-only e is

$$Z \geq 3kM$$

We distinguish four types of pairs (S_i, e), which are disjoint:

1. All accesses to e in $Sched$ are carried out in S_i, and e or a neighbor of e belong to the boundary of A.
2. One of the accesses to e in S_i is the first access to e in *Sched* (not counting type 1 pairs).
3. One of the accesses to e in S_i is the last access to e in *Sched* (not counting type 1 pairs).
4. e is accessed in S_i, but neither for the first nor for the last time.

Altogether, $Sched$ contains at most $4(N+1) + 4(N-1)$ pairs of type 1. We denote the total number of type 2, 3, and 4 pairs by Z^{in}, Z^{out}, and Z^{inOut}, respectively. From $Z^{in} + Z^{out} + Z^{inOut} \geq Z - 8N$ and $Z^{in} = Z^{out}$ follows $Z^{out} + Z^{inOut} \geq (Z - 8N)/2$.

At most M data are kept in cache in-between successive S_i's, summing up to $W \leq (k-1) \cdot M$ elements for the whole schedule. Consequently, at least

$$\begin{aligned} Z^{out} + Z^{inOut} - W &\geq 1.5kM - 4N - k \cdot M + M \\ &\geq 0.5\lfloor (N-1)^2/M^2 \rfloor \cdot M - 4N + M \end{aligned}$$

elements must be reloaded after having been replaced from cache. In other words, there are

$$0.5\lfloor (N-1)^2/M^2 \rfloor \cdot M - 4N + M = \Omega(N^2/M)$$

capacity misses, provided that N is significantly larger than M. □

To determine the constant factor in the bound, assume, for instance, $N \geq 100$, and $2 \leq M \leq N/10$. Then, $(N-1)^2/N^2 > 0.98$, and thus

$$\lfloor (N-1)^2/M^2 \rfloor \geq \lfloor 0.98N^2/M^2 \rfloor \geq \left(0.98N^2/M^2 - 1\right)$$

Since $N^2/M^2 \geq 100$, we can further estimate $0.98N^2/M^2 - 1 \geq 0.97N^2/M^2$. From $N^2/M \geq 10N$, we get $0.48N^2/M - 4N + M \geq 0.08N^2/M$ for the minimum number of cache misses. The constant factor approaches 0.5 as N^2/M grows.

3. UPPER BOUND

For $M < N$, the original codes take about $3N^2$ cache misses since each $A(I, J)$ is accessed in three columns. Tiling, a well-known compiler optimization [9, 10], improves the codes by exploiting temporal locality in the second dimension. The tiled code for Gauss-Seidel is given in Fig. 3. The Jacobi code is analogous and has been omitted for brevity. As Fig. 4 illustrates, tiling partitions A into horizontal stripes, which are dealt with in sequence, that is, first all updates of the uppermost stripe are carried out, then all updates of the next stripe, and so on. The stripes are denoted as tiles. Note that the data dependencies of Gauss-Seidel are respected, since iteration (I, J) is carried out after iterations $(I–1, J)$ and $(I, J-1)$.

Tile sizes are chosen such that three columns of a tile fit into cache simultaneously, that is, $S = \lfloor M/3 \rfloor$ for Gauss-Seidel. Capacity misses occur at tile boundaries only. Since the total length of these boundaries is $N \cdot (\lceil N/\lfloor M/3 \rfloor \rceil - 1)$, the number of capacity misses is

$$\begin{aligned} N \cdot (\lceil N/\lfloor M/3 \rfloor \rceil - 1) &\leq N^2/\lfloor M/3 \rfloor \leq N^2/(M/3 - 1) \\ &\leq 3N^2/(M-3) \leq 3.1N^2/M \end{aligned}$$

where the last inequality holds for $M \geq 100$.

The constant factor is somewhat higher for Jacobi since the cache must hold entries of two arrays. If the cache holds three columns

```
do  T = 1,  time
 do  II  = 1,  N-1,   S
  do  J = 1,  N-1
   do  I = II,  min(II+S-1,     N-1)
    A(I,J)   = W1*A(I-1,J)     + W2*A(I+1,J)
           + W3*A(I,J-1)     + W4*A(I,J+1)
           + W5*A(I,J+1)                    (*)
   end  do
  end  do
 end  do
end  do
```

Figure 3: Tiled code for Gauss-Seidel

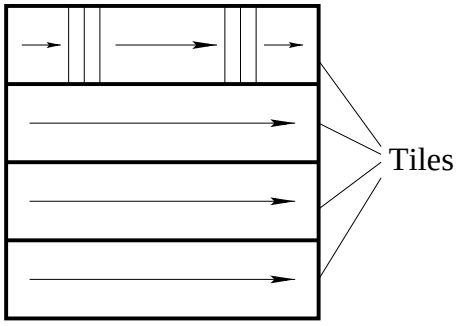

Figure 4: Execution order of the tiled scheme

of the right-hand array and one column of the left-hand array, we get up to $4.2N^2/M$ capacity misses.

The results indicate very close to optimum cache performance since the total number of cache misses, including cold misses, can be estimated by

$$\frac{\text{Upper bound}}{\text{Lower bound}} < \frac{(N+1)^2 - 4 + 3.1N^2/M}{(N+1)^2 - 4 + 0.08N^2/M} < 1 + 3.1/M$$

For the Jacobi scheme, the corresponding bound is $1 + 4.2/M$.

Since M is typically large, the results clearly discourage further research into temporal locality optimization. Nevertheless, for completeness, the following paragraphs discuss some modifications of tiling that bring the upper and lower bounds even closer together.

Improved Analysis

First, we can increase the tile size without changing the tiling technique itself. Observe that an update to $A(I, J)$ in Gauss-Seidel does not access $A(I-x, J-1)$ or $A(I+x, J+1)$ for $x > 0$. Thus, instead of three columns as assumed above, it is sufficient to keep two columns plus one element in cache. Similarly, the Jacobi scheme does not reuse elements of the left-hand array, and it is sufficient to keep two columns plus two elements in cache. This modification improves the relation between upper and lower bound

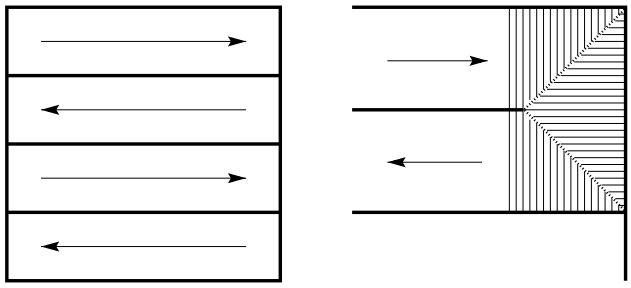

Figure 5: Snake-like scheme for Jacobi

to

$$\frac{(N+1)^2 - 4 + 2.1N^2/M}{(N+1)^2 - 4 + 0.08N^2/M} \leq 1 + 2.1/M$$

for both Gauss-Seidel and Jacobi.

In practice, the cache replacement policy must be taken into account. For matrix A, the LRU scheme works fine since the two columns to be kept in cache contain just the least recently used elements of A. In the Jacobi scheme, array B interferes with array A, but a write-around cache can avoid that.

Snake-Like Scheme for Jacobi

The codes of Fig. 3 do not reuse data between successive tiles, even though the tiles have data in common. Figure 5 depicts a snake-like scheme that exploits a certain amount of inter-tile reuse for Jacobi. The tiles have the same heights as above, namely (about) $M/2$, so that all intra-tile locality is exploited. Additionally, inter-tile locality is exploited at the turning points. The idea is illustrated in Fig. 5: Before a turning point, updates are carried out column-wise, and at the point, updates are carried out row-wise. While approaching the turning point, column lengths are reduced, and then they are increased again.

Assuming user-controlled data placement, cache size M is sufficient. The diagonal, which is marked by a dotted line in Fig. 5, has length $M/2$. At any one time, the cache must hold two reduced-length columns, as well as those entries of the diagonal that have been touched thus far. As the columns shrink when the diagonal grows, the elements fit into cache simultaneously.

Unfortunately, the performance impact of snaking is low, since inter-tile locality is exploited for only $M/2$ columns at the turning points. There is one turning point per tile (either on the left or on the right side), and thus the number of capacity misses is

$$2.1 \cdot (N - M/2) \cdot (N/M)$$

Snaking complicates the program structure, and thus impairs other performance factors such as the number of mispredicted branches and instruction cache misses. Therefore the overall performance is not improved.

Skewed Scheme for Gauss-Seidel

Snaking can be applied to the Jacobi scheme only. Since part of the rows are processed from right to left, the data dependencies of the Gauss-Seidel scheme are not respected. Figure 6 depicts a modification that is applicable to both schemes. It uses diagonal tiles instead of horizontal ones, and processes the tiles from left

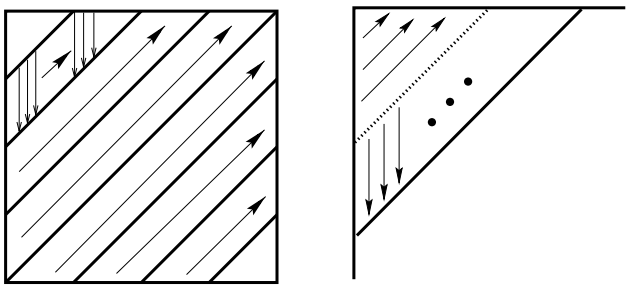

Figure 6: Skewed scheme for Jacobi

to right. Obviously, data dependencies are respected if, within a tile, the updates are carried out column-by-column, starting with the leftmost column.

Whether the scheme reduces cache misses depends on the particular values of N, M. The skewed scheme, like the base scheme, uses tile heights of about $M/2$, and capacity misses occur at tile boundaries. The sum of tile boundary lengths can easily be seen to be $2N^2/M$, which is the same as the total boundary length of the base scheme, indicating an equal number of cache misses.

The skewed scheme has the advantage that the heights of the first and last tiles may be somewhat increased, which is not possible for the base scheme: At the very beginning, data are updated in diagonals, instead of in columns, until the diagonal has reached length $M/2$ (dotted line in Fig. 6) These $M/2$ elements are kept in cache while a diagonal stripe of height $M/4$ is then updated column-by-column. Only thereafter, the first capacity misses occur, that is, the first (and last) tiles have width$3M/4$.

In the simulation of a small Gauss-Seidel instance with $N = 8$ and $M = 14$, our skewed scheme reduced the number of cache misses from 72 to 70. For realistic input sizes, the improvement is even less, and it is dominated by the higher costs that are due to a more complicated program structure.

4. RELATED WORK

The compiler optimization community has conducted much research on tiling (e.g. [3, 5, 9, 10]). Nevertheless, Douglas et al. [1] observe that current production compilers are not able to tile even elementary stencil codes. Part of the reason may be differences in the way tiling works for stencil codes as compared to dense linear algebra codes (the focus of compiler research). There, the data of a tile are moved into cache together, and then stay until the processing of the tile is complete. Tiled stencil codes, in contrast, keep only a few columns in cache, to be efficient.

Several papers deal with tiling specifically for stencil codes. Closest to ours is work by Rivera and Tseng [6] who present tiled 3D codes that are analogous to our 2D codes. They also discuss tile size selection and padding to avoid conflict misses. Like us, they refer to a single sweep and consider updates as a unit. The effectiveness of tiling is shown experimentally, lower bounds are not given.

In other papers, tiling is used to exploit reuse among different sweeps. Leiserson et al. [4] describe the base idea: The array is covered with kernel tiles of size $s \times s$. In order to carry out t sweeps on a kernel tile, a larger tile of size $s+2t \times s+2t$ is loaded into fast memory. In recent work, Douglas et al. [1], Sellappa [7], and Song and Li [8] adapt and improve the base scheme for specific cases.

Douglas et al. [1] and Sellappa [7] incorporate their modifications into multigrid programs. Multigrid is also considered in more theoretical work by Leiserson et al. [4] who show that cache misses can be asymptotically reduced if updates are not considered a unit.

5. CONCLUSIONS

This paper has derived matching upper and lower bounds on the number of cache misses for one sweep of the Jacobi and Gauss-Seidel relaxation methods. The bounds match up to a factor of about $1 + 2/M$, where M is the cache size. The result shows that the standard technique of tiling achieves a close to optimum number of cache misses.

Moreover we have investigated how the gap between upper and lower bounds can be closed. We found three modifications: increased tile size, snaking, and skewing that further reduce the number of cache misses. Snaking and skewing do not lead to overall speedups, however, as they complicate program structure. In ongoing work, we are generalizing our techniques to 3D stencil codes and tiling schemes for the time loop.

6. REFERENCES

[1] C. C. Douglas, U. Rüde, J. Hu, and M. Bittencourt. A guide to designing cache aware multigrid algorithms. In *Concepts of Numerical Software, Notes on Numerical Fluid Mechanics*. Vieweg-Verlag, 2001. To appear.

[2] J.-W. Hong and H. T. Kung. I/O complexity : The red-blue pebble game. In *Proc. ACM Symposium on Theory of Computing*, pages 326–333, 1981.

[3] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *Proc. ACM Int. Conf. on Supercomputing*, pages 482–491, 1999.

[4] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. *Journal of Computer and System Sciences*, 54(2):332–344, Apr. 1997.

[5] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *8th Int. Conf. on Compiler Construction*, pages 168–182. LNCS 1575, 1999.

[6] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In*Proc. Supercomputing*. IEEE, 2000. Available at http://www.supercomp.org/sc2000/Proceedings/start.htm.

[7] S. Sellappa. Cache-efficient multigrid algorithms. Master's thesis, University of North Carolina at Chapel Hill, Dept. of Computer Science, 2000.

[8] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 215–228, 1999.

[9] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, 1991.

[10] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 2000.