Electronic Transactions on Numerical Analysis.
Volume 10, 2000, pp. 21-40.

ISSN 1068-9613.

ETNA
Kent State University
etna@mcs.kent.edu

CACHE OPTIMIZATION FOR STRUCTURED AND UNSTRUCTURED GRID MULTIGRID*

CRAIG C. DOUGLAS†, JONATHAN HU‡, MARKUS KOWARSCHIK§, ULRICH RÜDE¶, AND CHRISTIAN WEISS‖

Abstract. Many current computer designs employ caches and a hierarchical memory architecture. The speed of a code depends on how well the cache structure is exploited. The number of cache misses provides a better measure for comparing algorithms than the number of multiplies.

In this paper, suitable blocking strategies for both structured and unstructured grids will be introduced. They improve the cache usage without changing the underlying algorithm. In particular, bitwise compatibility is guaranteed between the standard and the high performance implementations of the algorithms. This is illustrated by comparisons for various multigrid algorithms on a selection of different computers for problems in two and three dimensions.

The code restructuring can yield performance improvements of factors of 2-5. This allows the modified codes to achieve a much higher percentage of the peak performance of the CPU than is usually observed with standard implementations.

Key words. computer architectures, iterative algorithms, multigrid, high performance computing, cache.

AMS subject classifications. 65M55, 65N55, 65F10, 68-04, 65Y99.

1. Introduction. Years ago, knowing how many floating point multiplies were used in a given algorithm (or code) provided a good measure of the running time of a program. This was a good measure for comparing different algorithms to solve the same problem. This is no longer true.

Many current computer designs, including the node architecture of most parallel supercomputers, employ caches and a hierarchical memory architecture. Therefore the speed of a code (e.g., multigrid) depends increasingly on how well the cache structure is exploited. The number of cache misses provides a better measure for comparing algorithms than the number of multiplies. Unfortunately, estimating cache misses is difficult to model a priori and only somewhat easier to do a posteriori.

Typical multigrid applications are running on data sets much too large to fit into the caches. Thus, copies of the data that are once brought to the cache should be reused as often as possible. For multigrid, the possible number of reuses is always at least as great as the number of iterations of the smoother or rougher.

Tiling is an attractive method for improving data locality. Tiling is the process of decomposing a computation into smaller blocks and doing all of the computing in each block one at a time. In some cases, compilers can do this automatically. However, this is rarely the

*Received May 25, 1999. Accepted for publication February 1, 2000. Recommend by I. Yavneh. This research was supported in part by the Deutsche Forschungsgemeinschaft (project Ru 422/7-1), the National Science Foundation (grants DMS-9707040, ACR-9721388, and CCR-9902022), NATO (grant CRG 971574), and the National Computational Science Alliance (grant OCE980001N and utilized the NCSA SGI/Cray Origin2000).

†University of Kentucky, 325 McVey Hall - CCS, Lexington, KY 40506-0045, USA. (douglas@ccs.uky.edu)

‡University of Kentucky, Department of Mathematics, 715 Patterson Hall, Lexington, KY 40506-0027, USA. (jhu@ms.uky.edu)

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

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

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

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 [12].

We note that both the memory bandwidth (the maximum speed that blocks of data can be moved in a sustained manner) as well as memory latency (the time it takes move the first word(s) of data) contribute to the inability of codes to achieve anything close to peak performance. If latency were the only problem, then most numerical codes could be written to include prefetching commands in order to execute very close to the CPU's peak speed. Prefetching refers to a directive or manner of coding (which is unfortunately compiler and hardware dependent) for data to be brought into cache before the code would otherwise issue such a request. In general, a number of operations which may be using the memory bus can interfere with just modeling the cache by latency.

In this paper, suitable blocking strategies for both structured and unstructured grids will be introduced. They improve the cache usage without changing the underlying algorithm. In particular, bitwise compatibility is guaranteed between the standard and the high performance implementations of the algorithms. This is illustrated by comparisons for various multigrid algorithms on a selection of different computers for problems in two and three dimensions.

The code restructuring can yield a performance improvement by up to a factor of 5. This allows the modified codes to achieve a quite high percentage of the peak performance of the CPU, something that is rarely seen with standard implementations. For example, on a Digital Alpha 21164 processor based workstation, better than 600 out of a possible 1000 megaflops per second (MFlops/sec) has been achieved.

Consider solving the following set of problems:

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

where $u_i \in \mathbb{R}^{n_i}$. Each problem represents a linear system of equations to solve for a discretized partial differential equation on a grid Ω_i.

For any problem on Ω_i, a popular approximate solver is Gauss-Seidel with either the natural or red-black ordering. The red-black ordering has the advantage that it parallelizes in a nice way (communication reduces to sending half of the boundary information at a time which allows for overlapping communication and computing). On both serial and parallel computers it also reduces the cost of the prolongation procedure since only the black points need to be corrected. See [3], [4], [9], [11] for more details.

Consider the grid in Figure 1.1, where the boundary points are included in the grid. The usual red-black ordered Gauss-Seidel iteration performs Gauss-Seidel on all of the red points and then all of the black points. The algorithm can be translated into the following:

1. Update all of the red points in row 1.
2. Do $j = 2, N$
   - 2a. Update all of the red points in row j.
   - 2b. Update all of the black points in row $j - 1$.
   - 2c. End Do
3. Update all of the black points in row N.

When four grid rows of data (u_i and f_i) along with the information from the corresponding rows of the matrix A_i can be stored in cache simultaneously, this is a cache based algorithm.

The advantage is that all of the data and the matrix pass through cache only once instead of the usual twice. However, with substantial changes to the algorithm, almost all of the data

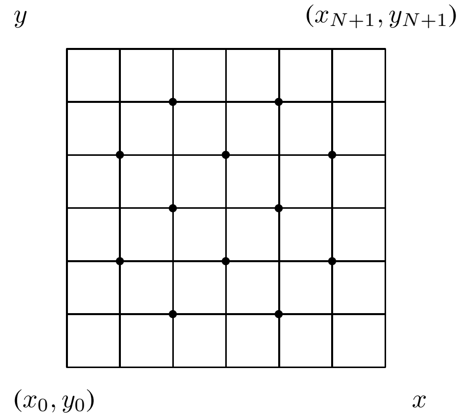


FIG. 1.1. *Simple grid with red points marked*

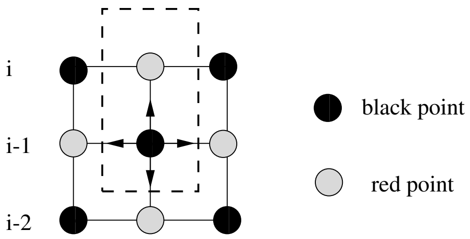


FIG. 2.1. *Data dependencies in a red-black Gauss-Seidel algorithm.*

and matrix passes through cache just once instead of $2m$ times for m iterations of red-black Gauss-Seidel. In addition, the new algorithm calculates bitwise the same answer as a non-cache Gauss-Seidel implementation. Hence, the standard convergence analysis still holds for the new algorithms. A technique to do this reduction can be found in [6].

Substantially better techniques to reduce the number of cache misses for Poisson's equation on tensor product grids can be found in §§2-3 for two and three dimensional problems.

A technique that is suitable for both two and three dimensional problems on unstructured grids can be found in §4.

Finally, in §5, we draw some conclusions and discuss future work.

2. Optimization techniques for two dimensions. The key idea behind data locality optimizations is to reorder the data accesses so that as few accesses as possible are performed between any two data references which refer to the same memory location. Hence, it is more likely that the data is not evicted from the cache and thus can be loaded from one of the caches instead of the main memory. The new access order is only correct, however, if no data dependency is violated during the reordering process. Therefore, the transformed program must yield results which are bitwise identical to those of the original program.

The data dependencies of the red-black Gauss-Seidel algorithm depend on the type of discretization which is used. If a 5 point stencil is used to approximate the differential operator and placed over one of the black nodes as shown in Figure 2.1, all of the red points that are required for relaxation are up to date, provided the red node above the black one is up to date. Consequently, we can update the red points in any row *i* and the black ones in row $i-1$ in a pairwise manner. This technique, which has already been mentioned in Section 1, is called *fusion technique*. It fuses two consecutive sweeps through the whole grid into a single sweep through the grid, updating both the red and the black nodes simultaneously.

This fusion technique applies only to one single red-black Gauss-Seidel sweep. If several successive red-black Gauss-Seidel iterations must be performed, the data in the cache is not reused from one sweep to the next, if the grid is too large to fit entirely into the cache. If we want to optimize the red-black Gauss-Seidel method further, we have to investigate the data dependencies between successive relaxation sweeps. Again, if a 5 point stencil is placed over one of the red nodes in line $i-2$, that node can be updated for the second time only if all of its neighboring black nodes have already been updated once. This condition is fulfilled as soon as the black node in line $i-1$ directly above the red node has been updated once. As described earlier, this black node may be updated as soon as the red node in line i directly above it has been updated for the first time. In general, we can update the red nodes in any two rows i and $i-2$ and the black nodes in the rows $i-1$ and $i-3$ in pairs. This *blocking technique* can obviously be generalized to more than two successive red-black Gauss-Seidel iterations by considering more than four rows of the mesh (see also [13]).

Both techniques described above require a certain number of rows to fit entirely into the cache. For the *fusion technique*, at least four adjacent rows of the grid must fit into the cache. For the *blocking technique*, it is necessary that the cache is large enough to hold at least $m*2+2$ rows of the grid, where m is the number of successive Gauss-Seidel steps to be blocked. Hence, these techniques can reduce the number of accesses to the highest level of the memory hierarchy into which the whole problem fits. These same techniques fail to utilize the higher levels of the memory hierarchy efficiently, especially the processor registers and the L1 cache, which may be rather small (e.g., 8 kilobytes on the Alpha 21164 chip). However, a high utilization of the registers and the L1 cache turns out to be crucial for the performance of our codes. Therefore, the idea is to introduce a two dimensional blocking strategy instead of just a one dimensional one. Data dependencies in the red-black Gauss-Seidel method make this more difficult than two dimensional blocking to matrix multiplication algorithms [2], for example.

The key idea for that technique is to move a small two dimensional *window* over the grid while updating all the nodes within its scope. In order to minimize the current working set of grid points, we choose the window to be shaped like a parallelogram (see Figure 2.2). The updates within reach of such a window can be performed in a linewise manner from top to bottom.

For example, consider the situation shown in Figure 2.2, which illustrates the algorithm for the case of two blocked Gauss-Seidel sweeps. We assume that a valid initial state has been set up beforehand, using a preprocessing step for handling the boundary regions of the grid. First consider the leftmost parallelogram (window). The red and the black points in the two lowest diagonals are updated for the first time, while the upper two diagonals remains untouched. Then the red points in the uppermost diagonal of the window are updated for the first time. As soon as this has been done, the black nodes in the next lower diagonal can also be updated for the first time. After that, the red and the black points belonging to the lower two diagonals are updated for the second time. Now consider the parallelogram on the right. We observe that the situation is exactly as it was initially for the left parallelogram: the red and the black nodes belonging to the lower two diagonals have already been updated once, while the two uppermost diagonals are still untouched. Consequently, we move the window to the position corresponding to the right parallelogram and repeat the update procedure. Generally speaking, the high utilization of the registers and the highest levels of the memory hierarchy, especially the processor registers and the L1 cache, is achieved by reusing the data in the overlapping region of the dashed line areas (see also [15, 14]). This overlap corresponds to

ETNA
Kent State University
etna@mcs.kent.edu

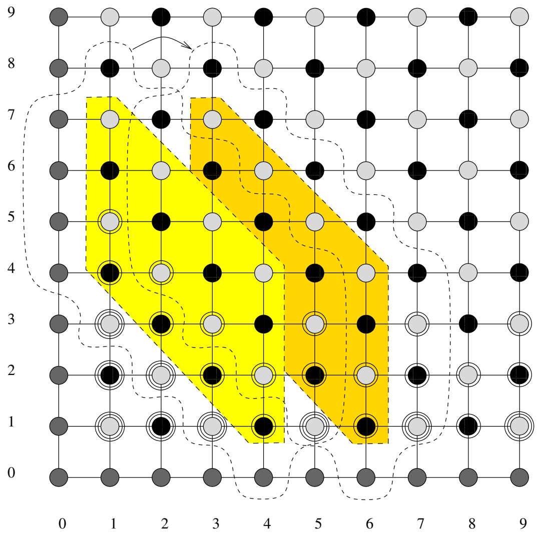


FIG. 2.2. *Two dimensional blocking technique for red-black Gauss-Seidel on a tensor product mesh.*

two successive window positions.

The previously described fusion and blocking techniques belong to the class of *data access transformations*. In order to obtain further speedups of the execution time, *data layout transformations*, like for example *array padding*, must also be taken into account and applied.

The term array padding refers to the idea of allocating more memory for the arrays than is necessary in order to avoid a high rate of *cache conflict misses*. A cache conflict miss occurs whenever two or more pieces of data which are needed simultaneously are mapped by the hardware to the same position within the cache (cache line). The higher the degree of associativity of the cache [7], the lower is the chance that the code performance suffers from cache conflict misses.

Figures 2.3 and 2.4 show the best possible performance which can be obtained for the red-black Gauss-Seidel algorithm alone and a complete multigrid V-cycle with four presmoothing and no postsmoothing steps applying the described data locality optimizations compared to a standard implementation. We implemented half injection as the restriction operator and linear interpolation as the prolongation operator. The performance results are shown both for a Digital PWS 500au with Digital UNIX V4.0D, which is based on the Alpha 21164 chip, and for a Compaq XP1000 with Digital UNIX V4.0E, which uses the successor chip Alpha 21264. Both machines run at a clock rate of 500 megahertz and have a theoretical floating point peak performance of 1 gigaflop each. The PWS 500au has 8 kilobytes of direct mapped L1 cache, whereas the XP1000 uses 64 kilobytes of two way set associative L1 cache and a faster memory hierarchy. Therefore, the speedups that can be achieved by applying the transformations described previously are higher for the Alpha 21164 based architecture. The performance on both machines increases with growing grid size until effects like register dependencies and branch missprediction are dominated by data cache miss stalls. Then, the performance repeatedly drops whenever the grid gets too large to fit completely into the L2 or L3 cache on the Digital PWS 500au, or the L1 and L2 cache on the Compaq XP1000.

In order to illustrate further the effectiveness of our optimization techniques we present a variety of profiling statistics for the Digital PWS 500au. Table 2.1 shows the percentages of data accesses that are satisfied by the individual levels of the memory hierarchy. These

ETNA
Kent State University
etna@mcs.kent.edu

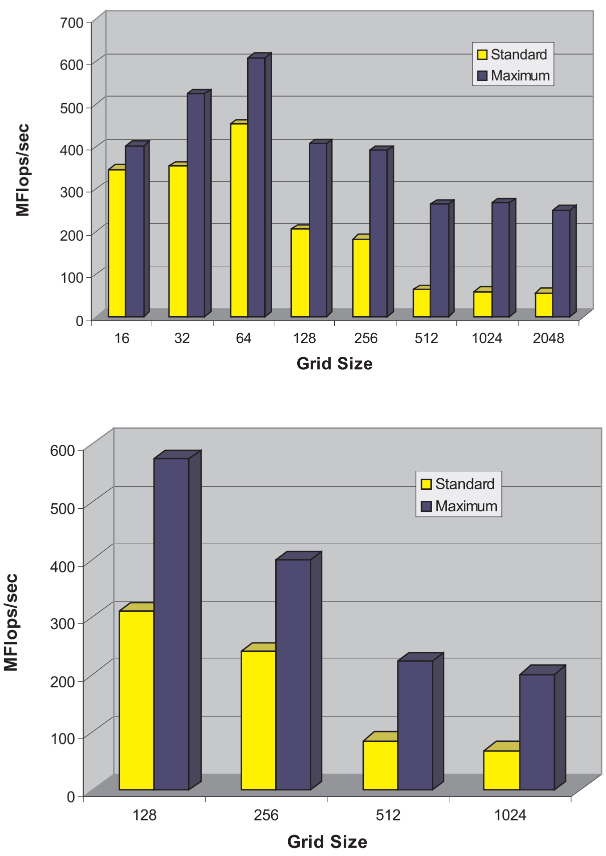


FIG. 2.3. *Speedups for the 2D red-black Gauss-Seidel method (above) and for 2D V(4,0)-multigrid cycles (below) on structured grids on a Digital PWS 500au.*

Relaxation Method	% of all accesses which are satisfied by				
	$\pm$	L1 Cache	L2 Cache	L3 Cache	Memory
Standard	5.1	27.8	50.0	9.9	7.2
Fusion	20.9	28.9	43.1	3.4	3.6
1D-Blocking (2)	21.1	29.1	43.6	4.4	1.8
1D-Blocking (3)	21.0	28.4	42.4	7.0	1.2
2D-Blocking (4)	36.7	25.1	6.7	10.6	20.9
2D-Blocking+Pad (4)	37.7	54.0	5.5	1.9	1.0

TABLE 2.1
Memory access behavior of different red-black Gauss-Seidel variants in two dimensions using a 1024×1024 *tensor product grid (on a Digital PWS 500au).*

statistics were obtained by using a low overhead profiling tool named *DCPI* [1]. *DCPI* uses hardware counters and a sampling approach to reduce the cost of profiling. In our case the slowdown for profiling was negligible. The numbers in parentheses denote how many successive Gauss-Seidel iterations are blocked into a single pass through the whole data set. The column "$\pm$" contains the differences between the theoretical and observed number of

ETNA
Kent State University
etna@mcs.kent.edu

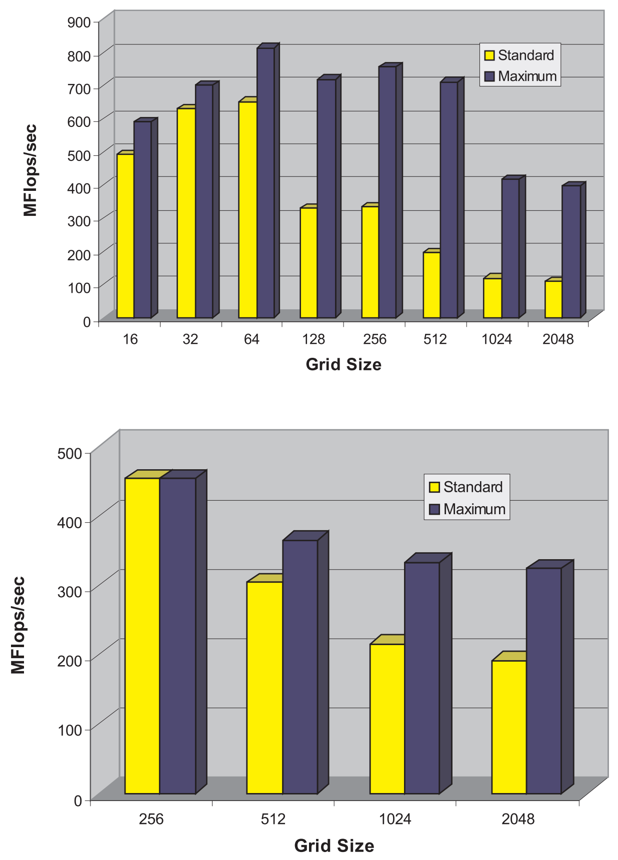


FIG. 2.4. *Speedups for the 2D red-black Gauss-Seidel method (above) and for 2D V(4,0)-multigrid cycles (below) on structured grids on a Compaq XP1000.*

load operations. Small entries in this column may be interpreted as measurement errors. Higher values, however, indicate that larger fractions of the array references are not realized as load/store operations by the compiler, but as very fast register accesses. In general, the higher the numbers in the columns "±", "L1 Cache" and "L2 Cache", the faster is the execution time of the code. Looking at the last two rows of Table 2.1, one can observe that array padding is particularly crucial for the L1 cache hit rate, which increases by more than a factor of 2 for the two dimensional blocking technique as soon as appropriate padding constants are introduced.

3. Optimization techniques for three dimensions. The performance results for a standard implementation of a red-black Gauss-Seidel smoother on a structured grid in three dimensions are comparable to the 2D case. The MFlops/sec rates drop dramatically on a wide range of currently available machines, especially for larger grids. Again, this is because data cannot be maintained in the cache between successive smoothing iterations.

To overcome this effect, we propose a three dimensional blocking technique, which is illustrated in Figure 3.1. This technique makes use of a small cube or a cuboid that is moved through the original large grid. According to the description of our optimization techniques for two dimensions in Section 2, this cuboid can be interpreted as a *three dimensional window*

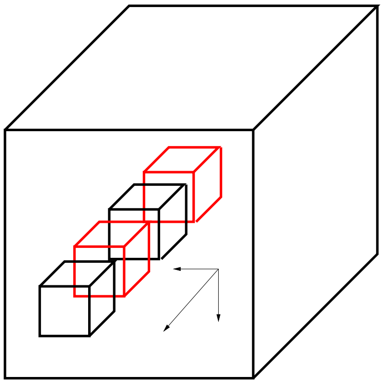


FIG. 3.1. *Three dimensional blocking technique for red-black Gauss-Seidel on a structured grid.*

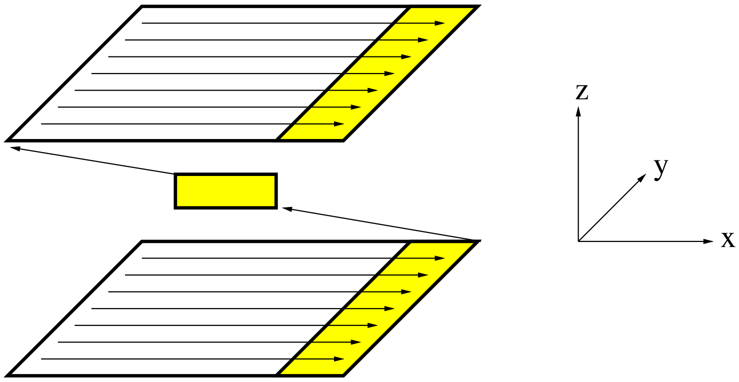


FIG. 3.2. *Array padding technique for three dimensional structured grids.*

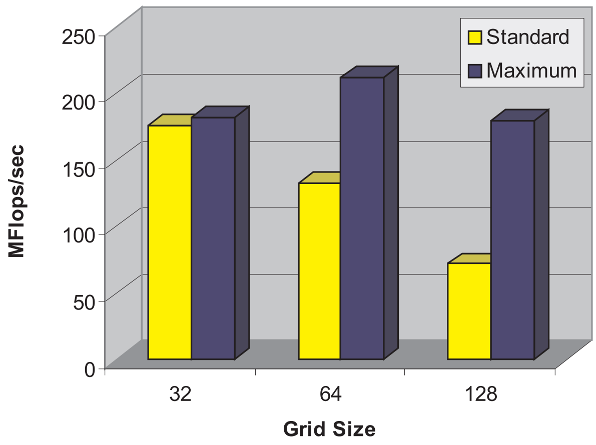
which denotes the grid nodes that are currently under consideration. In order that all the data dependencies of the red-black Gauss-Seidel method are still respected our algorithm has to be designed as follows.

After all the red points within the current position of the cuboid (window) have been updated, the cuboid has to be shifted back by one grid line in each dimension. Then the black points inside the new scope can be updated before the cuboid is again moved on to its next position. It is apparent that this algorithm incorporates the fusion and the blocking techniques, which have been described in more detail for the two dimensional case in Section 2. In addition, this algorithm is also suitable for blocking several Gauss-Seidel iterations. The four positions of the cuboid shown in Figure 3.1 illustrate that two successive Gauss-Seidel iterations have been blocked into one single pass through the entire grid.

However, this blocking technique by itself does not lead to significant speedups on the Alpha 21164 based PWS 500au, for example. A closer look at the cache statistics using the DCPI profiling tool (see Section 2) reveals that, again, the poor performance is caused by a high rate of cache conflict misses.

This problem can easily be pictured using the following model. We assume a three dimensional grid containing 64^3 double precision values which occupy 8 bytes of memory each. Furthermore, we assume an 8 kilobyte direct mapped cache (e.g., the L1 cache of the Alpha 21164). Consequently, every two grid points which are adjacent with regard to the trailing dimension of the three dimensional array are $64 \times 64 \times 8$ bytes away from each other in the address space. Note that this distance is a multiple of the cache size. Thus, every two grid nodes which satisfy this neighboring condition are mapped to the same cache line by the hardware and therefore cause each other to be evicted from the cache, in the end resulting in a very poor performance of the relaxation code.

Again, as in the two dimensional case, array padding turns out to be the appropriate data layout transformation technique to mitigate this effect. Figure 3.2 illustrates our padding technique. Firstly, we introduce padding in x direction in order to avoid cache conflict misses caused by grid points which are adjacent in dimension y. Secondly, we use padding to increase the distance between neighboring planes of the grid. This reduces the effect of z adjacent nodes causing cache conflicts. This kind of interplane padding is very crucial for code efficiency and has to be implemented carefully. In our codes this interplane padding is introduced by making use of both dexterous index arithmetic and the fact that Fortran compilers do not check any array boundaries to be crossed.



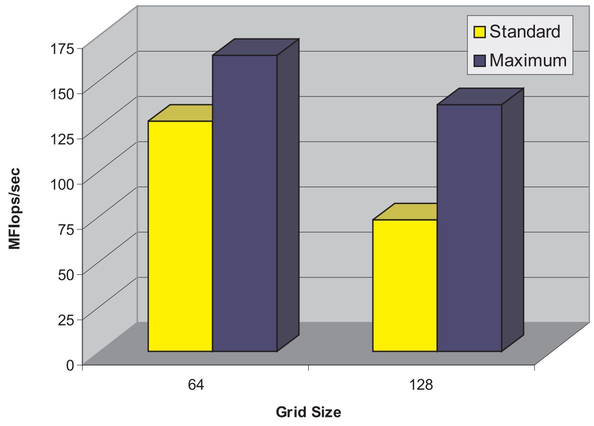
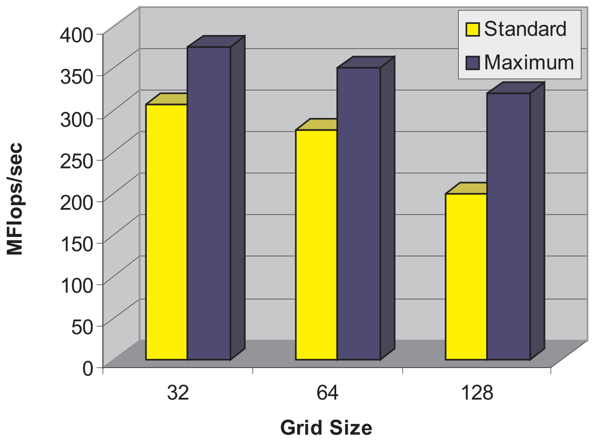


FIG. 3.3. *Speedups for the 3D red-black Gauss-Seidel method (above) and for 3D V(4,0)-multigrid cycles (below) on structured grids on a Digital PWS 500au.*

Figures 3.3 and 3.4 show the speedups that can be obtained on the Digital PWS 500au and on the Compaq XP1000 machines (see Section 2). We do not show speedup results for

ETNA
Kent State University
etna@mcs.kent.edu



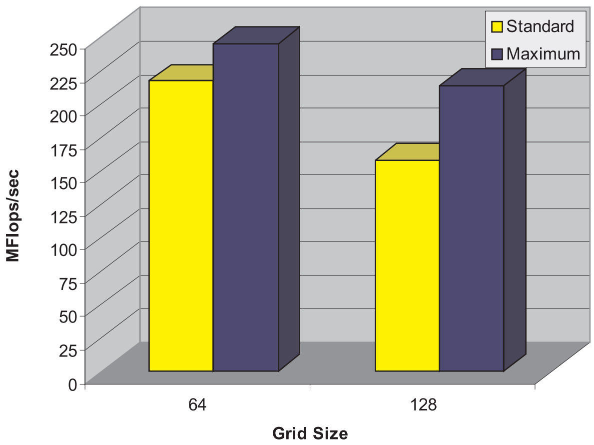


FIG. 3.4. *Speedups for the 3D red-black Gauss-Seidel method (above) and for 3D V(4,0)-multigrid cycles (below) on structured grids on a Compaq XP1000.*

smaller grids since they are negligible. This is due to the large ratio of the size of the cuboid to the size of the whole grid.

As we have already mentioned in Section 2 for the two dimensional case, the PWS 500au tends to be more sensitive to our optimization techniques. This is mainly due to the larger L1 cache, its higher associativity, and the lower memory access times of the XP1000 architecture.

The effectiveness of our code transformation techniques is of course influenced by several properties of the underlying machines and the compilers which we used. Nevertheless, the results we obtain are similar for a wide variety of machines and not just restricted to the examples presented in this paper.

4. Unstructured grids. A detailed cache analysis is much more difficult for unstructured grid problems. The problem coefficients are variable, and the number of connections varies from grid point to grid point. Furthermore, data storage must involve indirect addressing. Therefore, our cache optimizations for unstructured grids depend less on the technical details of the cache than in the case of structured grids. The goal is the same, however, as in the structured grid case. We wish to optimize the Gauss-Seidel smoother for cache while

Available cache size	Largest grid that fits in cache	Number of relaxations desired					
		3		4		5	
		%	#nodes	%	#nodes	%	#nodes
128 kb	48×49	80.44	1892	73.21	1722	66.33	1560
256 kb	68×69	85.93	4032	80.61	3782	75.45	3540
512 kb	97×97	89.96	8464	86.09	8100	82.30	7744
1024 kb	136×137	92.81	17292	90.01	16770	87.25	16256

TABLE 4.1
Percentage of nodes which are updated completely with one pass through cache.

maintaining bitwise the same solution as in standard Gauss-Seidel. The optimization, however, will be for larger cache sizes.

4.1. Motivation from Structured Grids. To understand what might be gained from cache optimization in the unstructured grid case, we first examine a structured grid problem. Assume a 5 point discretization on a rectangular mesh. (The same ideas apply to a 9 point stencil.) Suppose that an $m \times n$ block of nodes fits in cache. All nodes in cache get one update. We then shrink the block we are computing on by one along its boundary and relax again on an $(m-2) \times (n-2)$ subblock. Note that a buffer of one column and one row of nodes is not updated in order to maintain bitwise the same answer as the standard Gauss-Seidel. Continuing, on an $(m-2i) \times (n-2i)$ subblock, $i+1$ relaxations are performed while still maintaining the exact same updates as in the standard implementation of Gauss-Seidel. Assuming that all four sides of the block are cache block boundaries, $(m-2i) \times (n-2i)$ nodes get $i+1$ updates. We can say that in general $(m-2k+1) \times (n-2k+1)$ nodes get k updates. Hence, if we require only k updates,

$$\frac{(m-2k+1)(n-2k+1)}{mn} \times 100\%$$

of the data passes through cache once instead of k times.

For most multigrid applications k is quite small, typically 1 to 5. The case $k = 1$ is irrelevant to this paper. For many caches, m and n are large. Table 4.1 shows how many nodes are completely updated with one pass through cache for a variety of cache sizes and $k \in \{3, 4, 5\}$. We assume a 5 point discretization and that half of the total cache is available for data storage. We also assume that a data word is 8 bytes and that there is one unknown per node. For each unknown, the Gauss-Seidel algorithm requires the nonzero coefficients from a row of the matrix, the corresponding unknowns, and the right hand side entry.

4.2. Extension to Unstructured Grids. As in the structured case, we again are optimizing the smoother portion of the multigrid code. Our strategy involves several preprocessing steps, each of which will be discussed in detail:

- For each multigrid level, the physical grid is partitioned into contiguous blocks of nodes (*cache blocks*).
- For each cache block, sets of nodes which are a given distance from the cache block boundary are identified.
- Within each cache block, the nodes are renumbered according to this distance information.
- All matrices and operators are reordered based on the new nodal ordering.

After the preprocessing, we can perform cache aware Gauss-Seidel updates:

- Perform as many Gauss-Seidel updates as possible on each cache block without referencing data from other cache blocks.

ETNA
Kent State University
etna@mcs.kent.edu

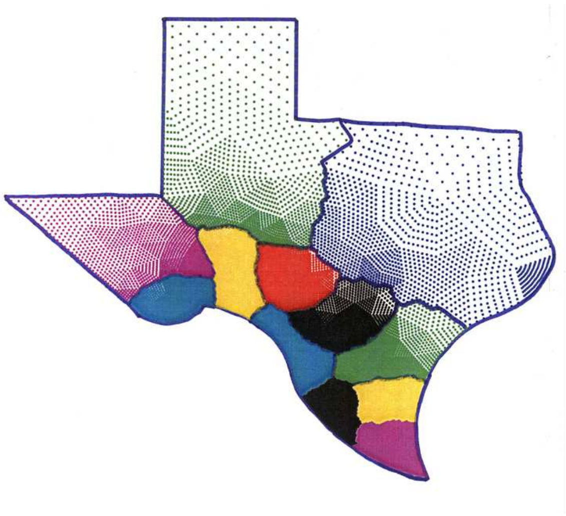


FIG. 4.1. *Decomposition of a grid into cache blocks by using the METIS load balancing package.*

- Calculate the residual wherever possible as Gauss-Seidel is performed.
- Revisit the cache blocks to finish updating nodes on the boundary of the cache blocks.

4.3. Decomposition of Grid. The first step of our strategy is to decompose the grid on each multigrid level into *cache blocks*. A cache block consists of nodes which are contiguous. 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 boundaries 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. The package which we use is the METIS library [8]. An example of a Texas shaped grid decomposed by METIS into twelve cache blocks is given in Figure 4.1.

4.4. Identification of Subblocks. 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 other subblocks. Identifying the number of updates possible for each node in a block without referencing another block is equivalent to identifying the distance of each node to the cache block boundary. To differentiate between the physical problem boundary and cache block boundaries, we label as $\partial\Omega_s$ the nodes in cache block Ω_s which depend on any node in another Ω_i.

Consider a cache block Ω_s. Let the vector D be such that D_i is the number of relaxations permissible on any node i in Ω_s. Then we see that D_i is the length of the shortest path between i and any node on the boundary $\partial\Omega_s$, where the length of a path is the number of nodes in a path. We assume that D_i = 1 for any node on $\partial\Omega_s$. A node is *marked* if its distance is known, and a node is *scanned* if it is marked and all of its neighbors' distances are known. Algorithms 1 and 2 calculate the distance of each node in a block. Algorithm 1 marks all nodes of distance one or two. Algorithm 2 finds the distances of all other nodes in a cache block. The motivation for both algorithms is Dijkstra's method for finding the shortest path between two nodes [5].

Algorithm 1 Mark cache boundary nodes.

Label-Boundary-Nodes

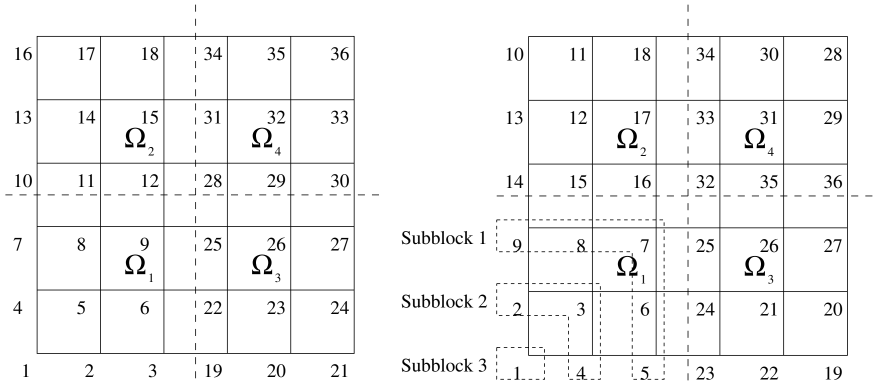
Initialize stacks S_1 and S_2.
Set distance $D_i = 0$ for all nodes i in Ω_s.
for each node i in Ω_s such that $D_i == 0$ **do**
  if i is on $\partial\Omega_s$ **then**
    Push i onto S_1.
    while S_1 is not empty **do**
      Pop node i off S_1.
      if i is in $\partial\Omega_s$ **then**
        Set $D_i = 1$.
        for each node j connected to i **do**
          if ($D_j == 0$) AND (j is in Ω_s) **then**
            Set $D_j = -1$.
            Push j onto S_1.
          end if
        end for
      else
        Set $D_i = 2$.
        Push i onto S_2.
      end if
    end while
  end if
end for

Algorithm 2 Mark cache interior nodes.

Label-Internal-Nodes

Set current subblock $s = 2$.
Set current distance $c = 2$.
Let m be the number of Gauss-Seidel updates desired.
while S_2 is not empty **do**
  if $s < m$ **then**
    Set current distance $c = c + 1$.
  end if
  Move contents of S_2 to S_1.
  while S_1 is not empty **do**
    Pop node i off S_1.
    for each node j adjacent to i **do**
      if $D_j == 0$ **then**
        Set $D_j = c$.
        Push j onto S_2.
      end if
    end for
  end while
  Let $s = s + 1$.
end while

ETNA
Kent State University
etna@mcs.kent.edu



(a) Before renumbering. (b) After renumbering.

FIG. 4.2. *Renumbering example.*

Algorithm 1 examines each node in a cache block until an unmarked boundary node is found. This node is pushed onto S_1 (lines 4-5). Each node i is now removed from S_1. If i is a cache boundary node, then i is marked as distance one, and all unmarked adjacent nodes j which are in Ω_s are marked as distance -1 (so that j will not be pushed onto S_1 more than once) and pushed onto S_1 for later scanning (lines 8-15). Otherwise, i is marked as distance two from the cache boundary and pushed onto stack S_2 (lines 16-18).

Each time S_1 is emptied, the FOR loop (line 3) iterates until an unmarked cache boundary node is found. The algorithm finishes when S_1 is empty and there are no more unmarked cache boundary nodes. At the conclusion of Algorithm 1, all cache boundary nodes have been marked as distance one, and all nodes which are distance two from the boundary have been marked and placed on S_2 in preparation for Algorithm 2.

The contents of S_2 are now moved to S_1. Algorithm 2 removes each node i from S_1 and scans it (lines 10-17). Each unmarked neighbor j of i is marked and pushed onto S_2. Once S_1 is empty, the stacks switch roles. Once m subblocks have been identified, all remaining unmarked nodes are marked as distance m from the cache boundary. Algorithm 2 continues until both stacks are empty. At the conclusion of Algorithm 2, vector D contains the smaller of m and the minimum distance from each node of block Ω_s to the boundary $\partial\Omega_s$.

4.5. Renumbering within Cache Blocks. We assume that the grid has been divided into k cache blocks and that within a block the numbering is contiguous (see Figure 4.2a). In block j the unknowns which are on the block boundary are marked as being on subblock 1, L_1^j. In general, let L_i^j denote those unknowns in block j which are distance i from the boundary. Let L_{n1}^1 be the subblock of block 1 farthest from $\partial\Omega_1$. The algorithm renumbers the unknowns in $L_{n1}^1, L_{n1-1}^1, \ldots, L_1^1, L_{n2}^2, \ldots L_1^2, \ldots, L_{nk}^k, L_{nk-1}^k, \ldots, L_1^k$, in that order (see Figure 4.2b). The result is a nodal ordering which is contiguous within blocks and subblocks. This ordering has the property that unknowns which are closer to the block boundary have a higher number than those further from the boundary. Numbering within a subblock is contiguous but arbitrary.

Assuming m updates, we find (where possible) $n1 = m + 1$ subblocks. Subblocks L_{n1}^1 and L_{n1-1}^1 are treated as one subblock when updates are made. The residual calculation, however, is different in each subblock. This will be discussed in more detail in §4.8.

ETNA
Kent State University
etna@mcs.kent.edu

As the final preprocessing step, all matrices and transfer operators are reordered according to the new grid numbering.

4.6. Computational Cost of Preprocessing. The decomposition of the grid into cache blocks, subblock identification, and the Gauss-Seidel algorithms are the only differences between our cache aware multigrid code and a standard multigrid code. The computational cost of subblock identification can be calculated by analyzing Algorithms 1 and 2. It can be shown that the combined complexity of the two algorithms on the finest grid is no greater than $\frac{7}{d^2}$ sweeps of standard Gauss-Seidel over the finest grid, where d is the number of degrees of freedom per node. This estimate is pessimistic. In experiments, the combined complexity is less than one sweep of standard Gauss-Seidel over the finest grid. (See §4.10.)

4.7. Updating within Cache Blocks. Once all matrix and grid operators have been reordered, the multigrid scheme can be applied. Assume that m smoothing steps are applied. Within cache block Ω_j, all nodes receive one update. All nodes in subblocks $L^j_{m+1}, \ldots, L^j_2$ are updated a second time. All nodes in subblocks $L^j_{m+1}, \ldots, L^j_3$ are updated a third time. This proceeds until all nodes in L^j_m and L^j_{m+1} have been updated $m-1$ times. Finally, all nodes in L^j_{m+1} and L^j_m are updated once more, a partial residual is calculated in L^j_m, and the entire residual is calculated in L^j_{m+1}.

4.8. Residual Calculation. 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. Consider the linear equation $Ax = b$, where $A = (a_{ij})$. Suppose that we solve this system by a Gauss-Seidel method. We would like to calculate the residual, $r = b - Au^k$, where u^k is the calculated solution after k updates. On the final Gauss-Seidel update (assuming m updates),

$$u_i^m = \frac{1}{a_{ii}}(b_i - \sum_{j<i} a_{ij} u_j^m - \sum_{j>i} a_{ij} u_j^{m-1}) \equiv \frac{1}{a_{ii}}(b_i - C_i^m - U_i^{m-1})$$

The partial sum C_i^m also appears in the residual calculation

$$r_i = b_i - \sum_j a_{ij} u_j^m = b_i - \sum_{j<i} a_{ij} u_j^m - a_{ii} u_i^m - \sum_{j>i} a_{ij} u_j^m = b_i - C_i^m - a_{ii} u_i^m - U_i^m.$$

Therefore we can save C_i^m and use it later in the residual calculation. Furthermore, $a_{ii} u_i^m = b_i - \sum_{j \neq i} a_{ij} u_j^m$. Hence, after the final Gauss-Seidel update on unknown x_i, only U_i^m need be calculated to finish the residual r_i. In our implementation, the partial sum U_i^m corresponds exactly to the unknowns x_j which are in the neighboring unfinished subblock.

These observations motivate our implementation of the residual calculation in the cache aware Gauss-Seidel routine. Consider the first pass through cache block Ω_s during a smoothing step. All unknowns are completely updated except those in subblocks $1, \ldots, m-1$ (assuming m updates). We assume that subblock j consists of nodes distance j from $\partial\Omega_s$. The key point is that each unknown x_i in subblock j depends only on unknowns in subblocks $j-1$, j, and $j+1$. Therefore, the residual can be fully calculated for all unknowns in subblock $m+1$ (if it exists). The partial sum C_i^m is available for the residual calculation at each unknown x_i in subblock m. No residual calculation is possible for unknowns in subblocks $1, \ldots, m-1$.

On the pth pass through Ω_s, the nodes are updated in order on subblocks $1, 2, \ldots, m-p+1$. Note that all nodes in subblock $m-p+1$ are now fully updated. This means that the partial sum C_i^m is available for the residual calculation for each unknown x_i in subblock

$n-p+1$ and the partial sum U_j^m is available for each unknown x_j in subblock $m-p+2$. In other words, as soon as the nodes in subblock $m-p+1$ are being updated for the last time, the residual calculation for subblock $m-p+1$ can be started, and the residual calculation for subblock $m-p+2$ can be finished.

For example, assume that 3 Gauss-Seidel updates are to be performed in a block with at least 3 subblocks. The residual can be fully calculated in cache for all nodes except those in subblocks 1-3. Furthermore, a partial residual can be calculated in subblock 3. On the second pass through the cache block, subblock 2 is fully updated, a partial residual is computed in subblock 2, and the residual is completed in subblock 3. On the third pass through the block, subblock 1 is fully updated. Once all cache blocks are fully updated, the residual can be completed in the first subblock of each block.

4.9. Revisiting Cache Blocks. As the problem size increases, the strategy for revisiting cache blocks to finish updating boundary unknowns (the backtracking scheme) becomes more important. In the worse case, boundary information for each block must pass through cache m times, where m is the number of relaxations desired. For a problem with a large number of cache blocks, a cache aware Gauss-Seidel implementation with a poor backtracking scheme might spend the majority of the computational time revisiting blocks. Hence, our objective is to develop a backtracking scheme which reduces the number of times that boundary information must pass through cache.

Suppose we have a decomposition of a grid into cache blocks. Each block can be thought of as a node of a graph. For any two adjacent blocks, the corresponding nodes will have an edge connecting them. This dual graph can be visualized as an adjacency matrix. By reducing the bandwidth of this matrix, we can potentially find an ordering for the cache blocks so that boundary information can be reused in cache during the backtracking phase of Gauss-Seidel. This permutation of the cache blocks is applied in the preprocessing phase. First, the cache blocks are reordered, then the subblocks are identified, the nodes are renumbered, and the operators are renumbered as previously described.

4.10. Numerical Experiments. All numerical experiments were performed on a SGI O2 with a 300 megahertz IP32 R12000 CPU, 128 megabytes of main memory, a 2×32 kilobyte split L1 cache, and a 1 megabyte unified 4 way secondary cache. Cache line lengths are 32 bytes (L1 instruction cache), 64 bytes (L1 data cache), and 128 bytes (L2 cache) [10]. The native cc and f77 compilers were used under IRIX 6.5. The METIS software package was used to decompose all domains into cache blocks. Unless otherwise stated, an effective cache size of 512 kilobytes is assumed.

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.

The first experiment compares three different Gauss-Seidel implementations. (See Table 4.2.) Note the decrease in speedup as the matrix order increases. The larger system has thirty-seven cache blocks, whereas the smaller system has nine. Our policy of revisiting the cache blocks in order may account for the worse performance in the larger system. Another possible cause is the worse condition of the cache blocks in the larger system. In the smaller system 69% to 86% of the nodes in each block can be updated when the block is first visited. For the larger system, however, only 58% to 72% of the nodes can be updated on the first visit to twenty-eight of the thirty-four cache blocks. For each system, the speedup increases with the number of updates. With more updates, more work is done in cache during the first visit to a block.

matrix order		Relaxations performed			
		2	3	4	5
31358	CAGSI	0.11	0.13	0.15	0.19
	GSI	0.23	0.30	0.38	0.46
	GSS	0.23	0.31	0.38	0.46
	speedup	2.09	2.31	2.53	2.42
124414	CAGSI	0.49	0.60	0.74	0.97
	GSI	1.03	1.35	1.74	2.04
	GSS	1.02	1.33	1.63	2.03
	speedup	2.08	2.22	2.20	2.07

CAGSI: cache Gauss-Seidel, integrated residual
GSI: non-cache Gauss-Seidel, integrated residual
GSS: non-cache Gauss-Seidel, separate residual

TABLE 4.2
CPU times (in seconds) from Gauss-Seidel updates with residual calculation on matrix arising from linear elastic problem on Texas shaped domain.

V cycles		# of updates on coarse/fine grids			
		2-2	3-3	4-4	5-5
1	CAGSI	1.56	1.94	2.24	2.61
	GSS	2.73	3.73	4.48	5.41
	GSI	2.78	3.77	4.54	5.45
	speedup	1.75	1.92	2.00	2.07

CAGSI: cache Gauss-Seidel, integrated residual
GSI: non-cache Gauss-Seidel, integrated residual
GSS: non-cache Gauss-Seidel, separate residual

TABLE 4.3
3 level V cycle applied to two dimensional linear elastic problem on domain shaped like the state of Texas. Matrix orders are 7966, 31358, and 124414, respectively.

The second numerical test compares cache aware and non-cache implementations of a three level V cycle. See Table 4.3 for speedup results. All three versions use the same grid operators. The grids are nested, and the two finer grids were created by uniform refinement. The matrices are order 7966, 31358, and 124414, respectively. Due to uniform refinement, the majority of rows have fourteen nonzeroes. Therefore a usable cache size of 512 kilobytes holds information for approximately 1820 nodes. In all cases, less than 20% of the CPU time was spent in interpolation and restriction operations.

In the third test, we use a unintrusive performance measurement tool which relies on hardware counters to analyze the cache efficiency of the various Gauss-Seidel implementations within a multigrid code (see Figure 4.3). Note that the only difference between the multigrid implementations is the smoother. Observe that the L2 cache hit rate for the cache aware multigrid method is approximately 25% to 30% higher than the standard implementations (see Figure 4.3a). As the number of updates increases, the hit rate increases. This is expected, since more updates mean more work in each cache block during the first visit to that block. Similarly, the average number of times a cache line is reused increases as the number of updates increases, for the same reason (see Figure 4.3b).

Finally, we compare CPU times for the reordering algorithms with the time for one Gauss-Seidel sweep over the finest grid. Table 4.4 compares the time to reorder all grids in the linear elasticity problem to one sweep of standard Gauss-Seidel. We see that the time

standard GS plus residual	standard GS	mesh renumbering
0.71	0.30	0.09

TABLE 4.4
CPU times (seconds) for Algorithms 1 and 2 to renumber all levels (83178 nodes total, two degrees of freedom per node) compared with one sweep of standard Gauss-Seidel on finest level.

to renumber the nodes is less than that for one standard Gauss-Seidel sweep on the finest level.

As noted in §1, there are hardware constraints which limit possible speedups. This is especially true in the unstructured grid case, where indirect addressing is necessary. Nevertheless, there may be room for further improvement in the unstructured case, as evidenced by the cache statistics in Figure 4.3(b).

5. Conclusions. In this paper, we have introduced a number of algorithms to solve elliptic boundary value problems using cache memories in a much more efficient manner than is usual. These algorithms preserve the same numeric solutions as the standard algorithms and codes do.

Bitwise compatibility is important since it allows us to guarantee that our codes are just as correct as standard codes, share the same numeric properties (roundoff and stability), and have the same convergence rates as usual algorithms. Additionally, there is the possibility in the future of building a software tool to generate the type of codes we developed here.

Algorithms have been investigated for problems on structured grids in two and three dimensions for Poisson's equation. Speedups are about a factor of $2 - 5$ on a variety of platforms. A quite high percentage of the peak performance of a processor can be achieved by doing loop unrolling and a two dimensional blocking technique.

Another set of algorithms have been investigated for unstructured grids and scalar or coupled general elliptic partial differential equations. Speedups are about a factor of 2. This technique extends trivially to parallel computers and higher dimensional problems. Using an active set technique, like in §§2-3, might lead to better speedups in the single processor situation, but does not extend to parallel computers in a straightforward manner like the algorithm in this paper. More investigation is needed to answer this speculation.

One of the goals of this joint research is a comprehensive library for solving partial differential equations on one or more processors that utilize caches in an efficient manner with little tuning required by the end user. While this is still a goal, studies like the one in this paper, will lead to a realization of this goal in the near future.

REFERENCES

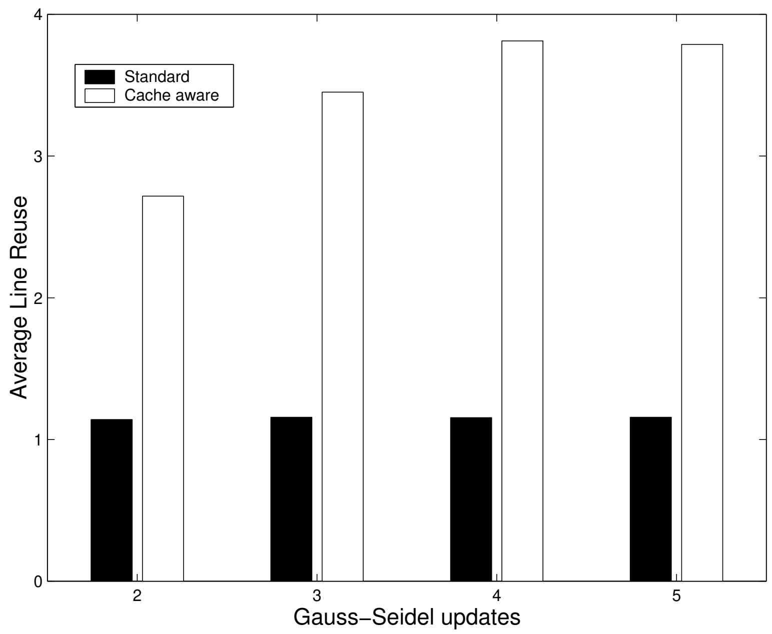
[1] J.M. ANDERSON, L.M. BERC, J. DEAN, S. GHEMAWAT, M.R. HENZINGER, S.A. LEUNG, R.L. SITES, M.T. VANDEVOORDE, C.A. WALDSPURGER, AND W.E. WEIHL, *Continuous profiling: where have all the cycles gone?*, in Proceedings of the 16th ACM Symposium on Operating system Principles, St. Malo, France, Oct. 1997.

[2] J. BILMES, K. ASANOVIC, C.-W. CHIN, AND J. DEMMEL, *Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology*, in Proceedings of International Conference on Supercomputing, July 1997.

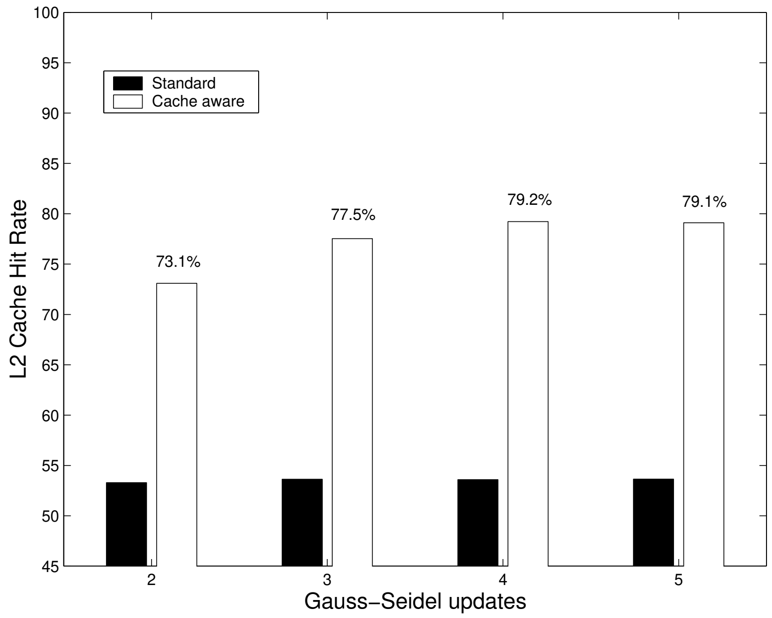
[3] W. L. BRIGGS, L. HART, S. F. MCCORMICK, AND D. QUINLAN, *Multigrid methods on a hypercube*, in Multigrid Methods: Theory, Applications, and Supercomputing, S. F. McCormick, ed., vol. 110 of Lecture Notes in Pure and Applied Mathematics, Marcel Dekker, New York, 1988, pp. 63–83.

[4] T. F. CHAN AND R. S. TUMINARO, *Design and implementation of parallel multigrid algorithms*, in Proceedings of the Third Copper Mountain Conference on Multigrid Methods, S. F. McCormick, ed., New York, 1987, Marcel Dekker, pp. 101–115.

[5] E. W. DIJKSTRA, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269–271.



(a) Average L2 Cache Line Reuse.



(b) L2 Cache Hit Rate.

FIG. 4.3. *Profiling statistics for multigrid codes using standard and cache aware smoothers.*

ETNA
Kent State University
etna@mcs.kent.edu

[6] C. C. DOUGLAS, *Caching in with multigrid algorithms: problems in two dimensions*, Paral. Alg. Appl., 9 (1996), pp. 195–204.
[7] J. HANDY, *The cache memory book*, Academic Press, New York, 1998.
[8] G. KARYPIS, *METIS serial graph partitioning and matrix ordering software*. In URL http://www-users.cs.umn.edu/∼karypis/metis/metis/main.shtml, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA.
[9] S. F. MCCORMICK, *Multigrid methods*, vol. 3 of Frontiers in Applied Mathematics, SIAM Books, Philadelphia, 1987.
[10] *MIPS R10000 microprocessor user's manual*, MIPS Technologies, Inc, version 1.1, 1996.
[11] S. V. PARTER, *Estimates for multigrid methods based on red–black Gauss–Seidel smoothings*, Numer. Math., 52 (1988), pp. 701–723.
[12] J. PHILBIN AND J. EDLER AND O. J. ANSHUS AND 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, Cambridge, MA, 1996, ACM, pp. 60–73.
[13] L. STALS, U. RÜDE, C. WEISS, AND H. HELLWAGNER, *Data local iterative methods for the efficient solution of partial differential equations*, in Proceedings of the Eighth Biennial Computational Techniques and Applications Conference, Adelaide, Australia, Sept. 1997.
[14] C. WEISS, W. KARL, M. KOWARSCHIK, AND U. RÜDE, *Memory characteristics of iterative methods*, in Proceedings of the Supercomputing Conference, Portland, Oreagan, Nov. 1999.
[15] C. WEISS, M. KOWARSCHIK, U. RÜDE, AND W. KARL, *Cache-aware multigrid methods for solving poisson's equation in two dimensions*. To appear in Computing, Springer Verlag, 2000.