

CACHE BASED MULTIGRID ON UNSTRUCTURED TWO DIMENSIONAL GRIDS

CRAIG C. DOUGLAS*, JONATHAN HU†, ULRICH RÜDE‡, AND MARCO BITTENCOURT§

1. Abstract. High speed cache memory is commonly used to address the disparity between the speed of a computer's central processing unit and the speed of a computer's main memory. It is advantageous to maximize the amount of time that data spends in cache. Tiling is a software technique which is often used to do just this. Tiling is not able, however, to handle dynamically changing data structures, such as those encountered in adaptively chosen, unstructured grids. We develop a variant of the Gauss-Seidel method for second order elliptic partial differential equations with variable coefficients. This variant keeps data in cache memory for much longer than non-cache implementations. As a result, our method is significantly faster than non-cache implementations. Examples from the structured grid case demonstrate the benefits of such a variant and provide motivation for the more difficult unstructured grid case. For this case, a publicly available load balancing package is used to decompose the grid into blocks of nodes which fit into cache. An $\mathcal{O}(n)$ algorithm is introduced that provides a one-time reordering of the nodes in each block. This new ordering permits significantly more Gauss-Seidel updates and residual calculation to be done in cache than in standard implementations. The cache aware Gauss-Seidel variant is incorporated into a multigrid code and tested by solving two dimensional linear elastic problems with multiple degrees of freedom per node. These experiments demonstrate cache hit rates and cache line reuse possible with a multigrid V cycle that incorporates such a cache aware method.

AMS Subject Classifications: 65M55, 65N55, 65F10, 65N30, 65F50, 65N50

Keywords: multigrid, unstructured grids, cache, tiling, iterative methods, domain decomposition, partial differential equations, software.

2. Introduction. In order to motivate the discussion, we first need a basic understanding of computer memory. The following description is based on [6] and [3]. A computer's central processing unit (CPU) performs the numerical and logical calculations when a program is executed. The data on which the CPU operates is stored in memory. When the CPU requires data which it does not already have, it makes a request to memory.

A modern CPU can perform a numerical operation much faster than memory can deliver data. Furthermore, the disparity between CPU and memory speed is growing. Therefore many hardware and software strategies have been developed to reduce the

*University of Kentucky, Departments of Mathematics and Mechanical Engineering and Center for Computational Sciences, 715 Patterson Office Tower, Lexington, KY 40506-0027, USA. E-mail: douglas@ccs.uky.edu.

†University of Kentucky, Department of Mathematics, 715 Patterson Office Tower, Lexington, KY 40506-0027, USA. E-mail: jhu@ms.uky.edu.

‡Lehrstuhl für Informatik X Universität Erlangen-Nürnberg, Martensstraße 3, D-91058 Erlangen, Germany. E-mail: rued@informatik.uni-erlangen.de.

§State University of Campinas, School of Mechanical Engineering, P.O. Box 6051, Campinas, SP 13083-970, Brazil. E-mail: mlb@fem.unicamp.br.

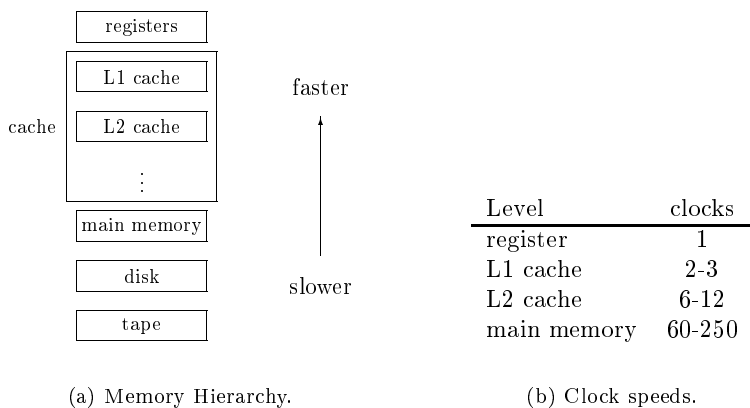


FIG. 2.1. Memory hierarchy and speeds [1].

time which the CPU is simply waiting for data. One common hardware strategy is to divide computer memory into a hierarchy of different layers, with the CPU linked directly to the highest level. (See Figure 2.1a.)

The highest level is the most expensive, smallest, and fastest of all layers. Lower levels are progressively larger, cheaper, and slower. The layers (from highest to lowest) are registers, cache, main memory, disk, and tape. By memory speed here, we mean time for the central processor to retrieve a piece of information from wherever it is stored. When the CPU makes a request for data, each layer (beginning with the first level cache) attempts to satisfy the request. The goal is that data which is frequently requested is as high in the hierarchy as possible and therefore available to the CPU with little delay.

The cache is itself a hierarchy L1, L2, ... (levels of cache). One or two levels of cache is typical. The L1 cache is smaller and 2 to 6 times faster than the L2 cache. The L2 cache in turn is smaller than but still 10 to 40 times faster than main memory (see Figure 2.1b). The smallest block of data moved in and out of a cache is a cache line. A cache line holds contiguous pieces of data, e.g., the components $v(i)$ through $v(i + j)$ of a vector, where j is small (e.g., 4, 16, or 32). If the data that the CPU requests is in a cache line, it is called a cache hit. Otherwise data must be copied from main memory into a line, resulting in a cache miss. It is clearly advantageous to maximize the number of cache hits, since the cache is faster than main memory in fulfilling the CPU's request. For more detailed information on memory hierarchies and cache design, see [4] and [8].

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.

For example, suppose that a code runs on a machine where the main memory has a latency of 100 clock cycles. We could preprocess the code so that new cache lines

are needed 100 cycles apart. Hence, we think that we have a code for which all data is available almost exactly when it is required. Unfortunately this is not how memory subsystems operate. The rest of the cache line may or may not be available once the first word is available. There may be more time needed to clear write back cache lines and the amount of time needed to read or write a cache line is frequently different (the write takes longer, as much as twice as long as a read). A number of operations may be using the memory bus that interfere with just modeling it by latency.

With a red-black Gauss-Seidel method and a problem too large to fit entirely into cache, each unknown passes through cache at least twice during each iteration. Rde and Stals [10],[11] and Douglas[3], however, use a simple change so that data passes through cache only once. They assume a rectangular grid and a 5 or 9 point stencil. Tiling is a software technique which is often used to maintain data in cache for as long as possible [9]. Tiling is not able to handle even the red-black case on a rectangular grid, much less dynamically changing data structures, such as those encountered in adaptively chosen, unstructured grids. Our goal is to develop a variant of the Gauss-Seidel method for second order elliptic partial differential equations with variable coefficients (see §4) on unstructured grids.

To be effective our technique needs a moderately large cache. On current processor types, the L1 cache is typically too small to be useful for our purposes. We therefore orient our technique towards the L2 cache, which is typically at least several hundred kilobytes. Note that on a system with a L3 cache, we target that instead of the much smaller L2 cache.

We assume that half of the L2 cache is available for program execution and that other processes (e.g., the operating system ones in particular) use the rest. This assumption came from a study of many different processors and operating systems while the research for [9] was being done.

3. Motivation from Structured Grids. Now 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 two columns and two rows of nodes is not updated in order to maintain bitwise the same answer as the standard (naturally ordered) 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 the typical current sizes of L2 caches, m and n are quite large. Table 3.1 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 1 unknown per node. For each

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 3.1

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

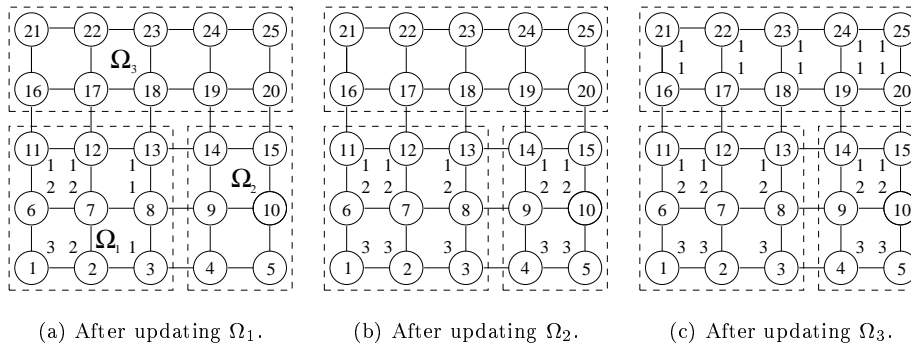


FIG. 3.1. Second structured grid example.

unknown, the Gauss-Seidel algorithm requires the nonzero coefficients from a row of the matrix, the corresponding unknowns, and the right hand side entry.

Now consider a second example involving a 5 point discretization on a rectangular mesh. Suppose we apply a Gauss-Seidel scheme. Assume that information for at least ten nodal values fits into cache. For clarity we use a domain decomposition notation even though we are not actually using a domain decomposition iterative method or developing a block preconditioner scheme. What follows is a global pointwise Gauss-Seidel scheme. We decompose the grid into three subdomains, Ω_1, Ω_2 , and Ω_3 (see Figure 3.1a). Each Ω_i is a block that fits entirely into cache.

Using the natural ordering, the nine nodes in Ω_1 are updated once. The key observation here is that a node can be updated as soon as all of its neighbors have the correct number of updates. Hence, we update nodes 1,2,6 and 7 a second time. Finally, node 1 is updated a third time. So without any data movement, we have done five more node updates than one standard Gauss-Seidel sweep would on these nine nodes. Furthermore, we have bitwise the same solution at node 1 as with the standard Gauss-Seidel algorithm has after three sweeps (see Figure 3.1a).

We now iterate in Ω_2 . We update all nodes one time. Then we return to Ω_1 and update nodes 3 and 8 a second time. Now we update nodes 4,5,9 and 10 a second time. Finally, we update nodes 2-5 a third time. This is depicted in Figure 3.1b.

Finally, we iterate in Ω_3 . We update all nodes in Ω_3 once (see Figure 3.1c). Returning to Ω_1 and Ω_2 , we update (in order) nodes 11-15 a second time, nodes 6-10 a third time, nodes 16-20 a second time, 21-25 a second time, 11-15 a third time, 16-20 a third time, and 21-25 a third time. The updating of all nodes is now complete.

Note that partially updated cache blocks must be partially revisited, but that only those nodes close to the boundaries between blocks need to be updated. In realistic situations the cache blocks are quite large with respect to the number of unknowns that have to be revisited.

From this small example, we see that with some rearrangement, bit-wise the same answer is calculated as a standard Gauss-Seidel implementation, but with less data movement through cache. In light of the discussion on memory hierarchies, we expect that such a rearrangement would result in faster run times than a standard Gauss-Seidel implementation on the same problem.

Note that the natural ordering was used in the small example. Suppose a much larger example had been considered on an irregularly shaped domain with irregularly shaped cache blocks. If we pick an ordering carefully, more updates are possible within a subdomain without referencing nodes in other subdomains. Identifying better orderings is the motivation for the following discussions.

4. Unstructured Grids. In this section, we extend the ideas from §3 to the case of an unstructured grid. We use the following partial differential equation as our model problem:

$$\begin{cases} -\nabla \cdot (a\nabla u) + su & = f \text{ in } \Omega, \\ u & = g_0 \text{ on } \Gamma_0, \\ u & = g_1 \text{ on } \Gamma_1, \end{cases}$$

where $\Gamma_0 \cup \Gamma_1 = \partial\Omega$, $\Gamma_0 \cap \Gamma_1 = \emptyset$, $a(x, y) \geq a_0 > 0$ for some $a_0 \in \mathbb{R}$, and $s(x, y) \geq 0 \forall (x, y) \in \Omega$. Ω is a simply connected open region in \mathbb{R}^2 . Our code allows for vector base functions.

Although we use structured grid examples in the following discussions, we treat the problems as variable coefficient problems and do not use any structured grid information. Once again, we modify the standard implementation of Gauss-Seidel to be cache aware.

First we identify blocks of nodes which fit into cache (cache blocks). To differentiate between the physical problem boundary and internal cache block boundaries, we label as $\partial\Omega_i$ the nodes in Ω_i which are adjacent to any node in another Ω_j . Then we relax as many times as possible on each block while it is in cache. Nodes on the boundary of each block potentially have dependencies on nodes in other blocks, so passing all of a block through cache once is not possible. If we minimize the boundaries of blocks, however, we can potentially reduce the dependency of one block on others. In particular we can design an algorithm which only brings a small number of cache lines into cache more than once.

Once a cache block is identified, we need to know how many relaxations are possible for each node without referencing another block. This requires calculating the minimum distance of all nodes to their cache block boundary $\partial\Omega_i$.

Consider a cache block Ω_j . Identifying the number of updates possible for each node in Ω_j without having to update a node in another block is equivalent to identifying the distance of each node to the boundary $\partial\Omega_j$. Let the vector D be such that D_i is the number of relaxations permissible on any node i in Ω_j . Then we see that D_i is the length of the shortest path between node i and any node on the boundary $\partial\Omega_j$, where the length of a path is the number of nodes in a path. We assume that $D_i = 1$ for any node i on $\partial\Omega_j$. The algorithms for calculating D for all nodes in a block are given in Figure 4.1. Algorithm “Label-Boundary-Nodes” finds distances of all nodes in the internal cache boundary. Algorithm “Label-Internal-Nodes” finds the distances

Algorithm Label-Boundary-Nodes:

```

Initialize stacks  $S_1$  and  $S_2$ .
Set  $D_i = 0$  for all  $i$ .
Find any node  $i$  on internal cache
boundary  $\partial\Omega$  of cache block  $\Omega$ .
Push  $i$  onto  $S_1$ .
Repeat until  $S_1$  is empty:
  Pop node  $i$  off  $S_1$ .
  If distance  $D_i$  is unknown:
    Set distance  $D_i = 1$ .
    For each node  $j$  adjacent to  $i$ 
    and in  $\Omega$ :
      If  $j$  is on  $\partial\Omega$ :
        Set distance  $D_j = 1$ .
        Push  $j$  onto  $S_1$ .
      else:
        distance  $D_j = 2$ .
        Push  $j$  onto  $S_2$ .
    End if
  End for
End if
End repeat

```

Algorithm Label-Internal-Nodes:

```

Repeat until  $S_2$  is empty:
  Move contents of  $S_2$  to  $S_1$ .
  Repeat until  $S_1$  is empty:
    Pop node  $i$  off  $S_1$ .
    For each node  $j$  adjacent to  $i$ 
    and in  $\Omega$ :
      If distance  $D_j$  is unknown:
        Set distance  $D_j = D_i + 1$ .
        Push  $j$  onto  $S_2$ .
      End if
    End for
  End repeat
End repeat

```

FIG. 4.1. Algorithms for finding the number of relaxations permissible on each node of a cache block.

of all nodes internal to a cache block. The motivation for both algorithms is Dijkstra’s method for finding the shortest path between two nodes [2]. A node is “marked” if its distance is known, and a node is “scanned” if all its neighbors’ distances are known. Note that scanned implies marked, but marked does not imply scanned.

Both algorithms maintain two stacks. Initially the first stack contains nodes that are marked but not scanned, i.e., their distances are known, but they may have neighbors whose distances are unknown. The second stack is initially empty. When a node is popped off of the first stack and scanned, all neighboring unscanned nodes are pushed onto the second stack. As soon as the first stack is empty, the stacks switch roles. The algorithm is done when both stacks are empty.

At the conclusion of Algorithm 4.1, vector D contains the minimum distance from each node of block Ω_j to the boundary $\partial\Omega_j$.

THEOREM 4.1. *The computational complexity of the combined algorithms “Label-Boundary-Nodes” and “Label-Internal-Nodes” is $\mathcal{O}(N)$, where N is the number of nodes in a cache block.*

Proof. Let N be the number of nodes in a cache block and c_i the number of connections to node i . Each node i is scanned only once, and at most c_i nodes are marked when i is scanned. Therefore $\sum_N c_i$ nodes are examined. We assume that $c_i \leq K$ for some $K \in \mathbb{N}$ and $K \ll N$, hence $\sum_N c_i \leq NK$, which is linear in N . \square

5. Description of implementation. Algorithms “Label-Boundary-Nodes” and “Label-Internal-Nodes” are implemented in C, as are other preprocessing steps. C allows flexibility in data structure design and dynamic memory allocation. The key Gauss-Seidel methods and the grid operators are implemented in Fortran 77 because Fortran compilers are well-optimized for numerical routines.

The implementation has several major blocks.

- For each block, perform one-time calculation of distances of each node from block boundary.
- For each block, perform one-time renumbering of nodes based on distance

information.

- Perform one-time reordering of matrix and operators based on new nodal ordering.
- Perform Gauss-Seidel relaxation on cache blocks.
- Calculate residual as soon as possible during Gauss-Seidel updates.

The distance calculation has already been discussed. We concentrate on the one-time reordering, the Gauss-Seidel method, and the residual calculation in §§5.1-5.3.

5.1. One-time Reordering Scheme. Domain decomposition software for parallel computers is readily available for identifying connected groups of nodes which minimize communication between groups. One such package is the METIS library [5]. Therefore we assume that the grid has been divided into k blocks of unknowns (cache blocks) and that within a block the numbering is contiguous (see Figure 5.1a). Note that contiguous numbering greatly simplifies the search for boundary nodes. In block j the unknowns which are on the block boundary are labeled 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 $L_{n_1}^1$ be the subblock of block 1 farthest from $\partial\Omega_1$. The algorithm renumbers the unknowns in $L_{n_1}^1, L_{n_1-1}^1, \dots, L_1^1, L_{n_2}^2, \dots, L_1^2, \dots, L_{n_k}^k, L_{n_k-1}^k, \dots, L_1^k$, in that order (see Figure 5.1b). 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.

Suppose we do m updates plus the residual calculation. Then it is necessary to have $m + 1$ subblocks instead of the m subblocks necessary for just the m updates. The first time cache block i is visited, all nodes in L_{m+1}^i and L_m^i can be fully updated. The residual can only be calculated in L_{m+1}^i since the nodes in L_m^i depend on nodes which are not yet fully updated. The residual calculation will be discussed in more detail in §5.2.

As a preprocessing step, all matrices and transfer operators are reordered according to the new grid numbering. A cache aware Gauss-Seidel method is then applied which is similar to that described in the structured grid case. We have chosen to visit the blocks sequentially. Hence, the cache aware Gauss-Seidel method updates the nodes in blocks 1,2,3, etc. as much as possible in that order. Then blocks 1,2,3, etc. are revisited in that order and updated as much as possible. The process continues until all unknowns are fully updated.

5.2. 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, $r = b - Au$, where u is the calculated solution. Let x be the approximation to u on the next to last iteration of the Gauss-Seidel iteration. On the final Gauss-Seidel update,

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

The partial sum C_i also appears in the residual calculation

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

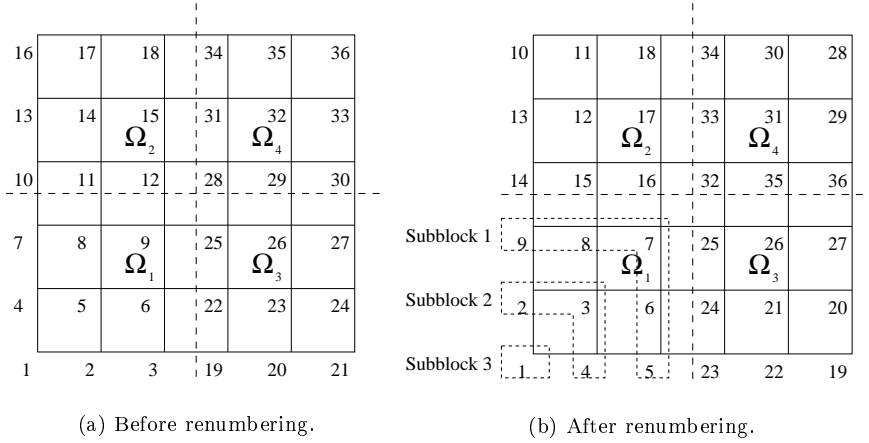


FIG. 5.1. *Renumbering example.*

Therefore we can save C_i and use it later in the residual calculation. Furthermore, $a_{ii}u_i = b_i - \sum_{j \neq i} a_{ij}x_j$. Hence, after the final Gauss-Seidel update on unknown x_i , only U_i need be calculated to finish the residual r_i . In our implementation, the partial sum U_i 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 a given cache block during a smoothing step. All unknowns are completely updated except those in subblocks $1, \dots, n-1$ (assuming n updates). We assume that block i consists of nodes distance i from $\partial\Omega_i$. 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 $n+1$ (if it exists). The partial sum C_i is available for the residual calculation at each unknown x_i in subblock n . No residual calculation is possible for unknowns in subblocks $1, \dots, n-1$.

On the p th pass through the same cache block, the nodes are updated in order on subblocks $1, 2, \dots, n-p+1$. Note that all nodes in subblock $n-p+1$ are now fully updated. This means that the partial sum C_i is available for the residual calculation for each unknown x_i in subblock $n-p+1$ and the partial sum U_j is available for each unknown x_j in subblock $n-p+2$. In other words, as soon as the nodes in subblock $n-p+1$ have been updated for the last time, the residual calculation for subblock $n-p+1$ can be started, and the residual calculation for subblock $n-p+2$ 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.

5.3. Data Structures. The cache aware Gauss-Seidel subroutine depends on 3 major data structures which store the matrix, starting and stopping indices for subblocks and blocks, and groups of rows with the same number of nonzeros. Each of the structures has been organized to enhance data locality.

The matrix is stored in a format requiring 3 arrays. The first array is a vector of indices into the nonzero matrix entries and the corresponding column indices. In addition to nonzero entries, the diagonal's reciprocal and the corresponding right hand side entry are also stored.

For a given cache block, all subblock information is stored contiguously. Subblock 1 is stored first, then subblock 2, and so on. Hence, looping over subblocks in the cache aware Gauss-Seidel implementation only requires looping over the same stored data.

The next major data structure is a two dimensional integer array which identifies groups of rows to which a given stencil applies. This is useful because the majority of rows have a common number of nonzeros. The use of a stencil eliminates loop overhead and facilitates software pipelining. Furthermore, no "if-then" checks are necessary in the innermost Gauss-Seidel loop, since groups of rows which can use the stencil have been identified in a preprocessing step. Finally, this idea is easily extendible to the cases where more than one type of stencil commonly occurs.

The final major data structure is a two-dimensional integer array which stores starting and stopping unknowns for each subblock and block. The major advantage of this data structure is that subblock information for a given cache block is stored contiguously in a very small array.

6. Numerical Experiments. Numerical experiments were performed on an SGI O2 with an 155 Mhz IP32 R10000 CPU, 128 MB of main memory, a 2×32 KB split L1 cache, and a 1 MB unified 4-way secondary cache. Cache line lengths are 32 bytes (L1 instruction cache), 64 bytes (L1 data cache), and 128 bytes (L2 cache) [7]. The native cc and f77 compilers were used. The METIS software package was used to decompose all domains into cache blocks. An effective cache size of 512 KB is assumed.

We solve a two dimensional linear elastic problem on a domain shaped like the state of Texas (see Figure 6.1). The domain is discretized with linear triangular elements, and each node has two degrees of freedom. The northern-most 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.

In the first test, we use a non-intrusive performance measurement tool to analyze the cache efficiency of the various Gauss-Seidel implementations. We solve by the Gauss-Seidel method in order to evaluate cache efficiency. Note that the L2 cache hit rate improves from about 50% for the very good, but standard implementation to 75% or more for the cache aware version (see Figure 6.2a). 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. The cache aware method also reuses cache lines 3 to 4 times on average. This number increases as the number of updates increases, for the same reason that the cache hit rate increases (see Figure 6.2b).

In the second test, we analyze the cache efficiency of a multigrid code using the various Gauss-Seidel implementations. We used 150 V cycles and three nested meshes to solve each problem. The multigrid code which uses a standard Gauss-Seidel implementation exhibits cache hit rates of about 50%. The hit rate improves to about

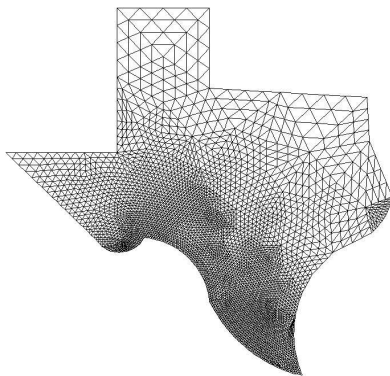


FIG. 6.1. *Coarsest grid used in numerical tests.*

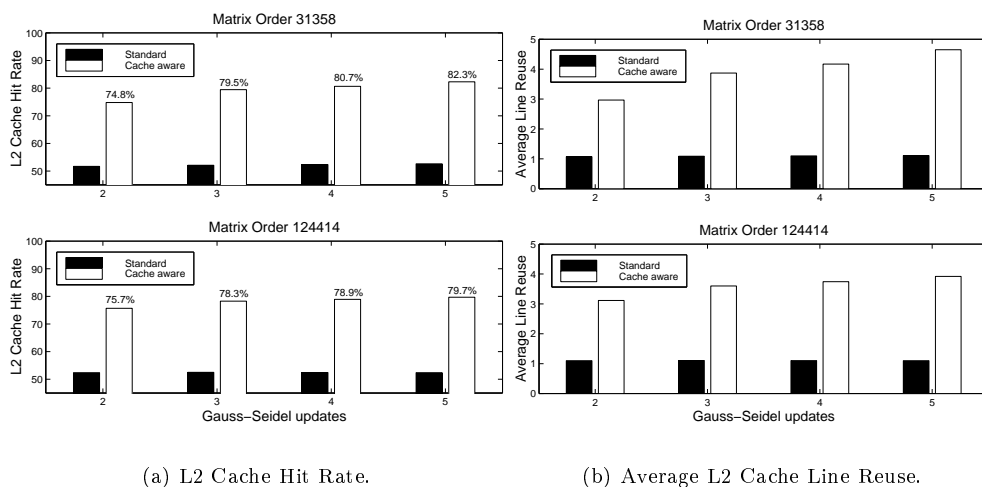


FIG. 6.2. *Profiling statistics for various Gauss-Seidel implementations only.*

70% or more for the multigrid code using cache aware Gauss-Seidel (see Figure 6.3a). The cache aware multigrid code also reuses cache lines about 2 to 3 times (see Figure 6.3b).

7. Concluding Remarks. We have implemented a Gauss-Seidel method that takes advantage of large and fast L2 caches. This implementation is designed for variable coefficient, multi-component PDE's on unstructured meshes. The key idea is a one-time reordering of the linear systems based on physical grid information and usable cache size. Testing indicates that a multigrid code which incorporates this cache aware method significantly improves the L2 cache usage. Furthermore, comparisons indicate that the Gauss-Seidel cache efficiency gives a good indication of the overall cache efficiency of the multigrid code.

We have not targeted integrated L1 caches since almost all of these are currently

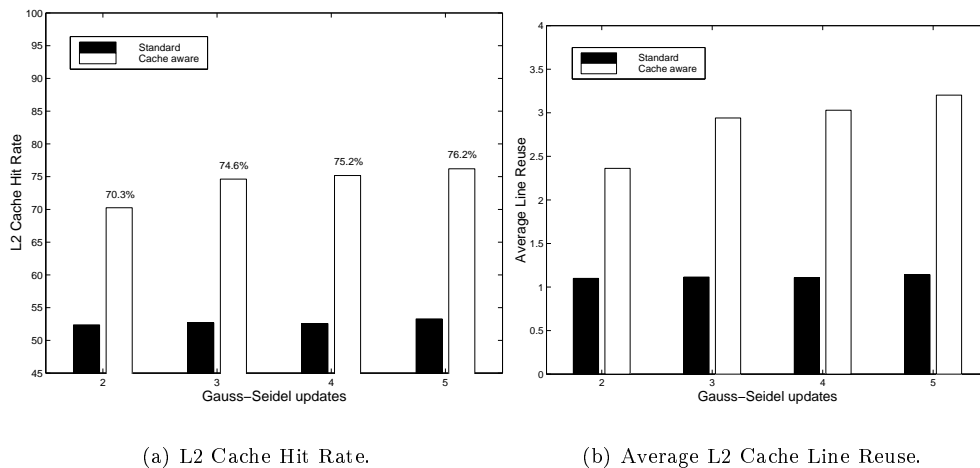


FIG. 6.3. Profiling statistics for multigrid codes.

too small. With the introduction of a new generation of CPU's with a very large on chip L1 cache (e.g., the HP PA8500 CPU with 1.5MB of L1 cache and no L2 cache) we will be able to extend our work to L1 caches. We expect to see a significant performance improvement. We also intend to extend our code to three dimensions.

REFERENCES

- [1] G. ASTFALK. Private communication, 1998.
- [2] E. W. DIJKSTRA, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269–271.
- [3] C. C. DOUGLAS, *Caching in with multigrid algorithms: problems in two dimensions*, Paral. Alg. Appl., 9 (1996), pp. 195–204.
- [4] J. HANDY, *The Cache Memory Book*, Academic press, New York, 1998. Second edition.
- [5] G. KARYPIS, *METIS serial graph partitioning and matrix ordering software*. In URL <http://www-users.cs.umn.edu/~karypis/memis/memis/main.shtml>, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA.
- [6] V. MEISER, ed., *Computational Science Education Project*, Computational Science Education Project, 1996. In URL <http://compsci.cas.vanderbilt.edu/csep/CSEP.html>.
- [7] *MIPS R10000 Microprocessor User's Manual*, MIPS Technologies, Inc, version 1.1 ed., 1996.
- [8] D. A. PATTERSON AND J. L. HENNESSY, *Computer Architecture: A Quantative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.
- [9] 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, Cambridge, MA, 1996, ACM, pp. 60–73.
- [10] L. STALS AND U. RÜDE, *Efficient implementation of multigrid on cache based architectures*. Submitted, 1997.
- [11] ———, *Techniques for improving the data locality of iterative methods*, Tech. Rep. MRR 038-97, Australian National University, 1997.