Portable memory hierarchy techniques for PDE solvers, Part II

Craig C. Douglas
Gundolf Haase
Jonathan Hu
Markus Kowarschik
Ulrich Rüde
Christian Weiss

The first part of this article, which appeared in the previous issue of *SIAM News*, detailed the architecture and behavior of microprocessor caches. It was also made apparent the impact that caches have on the performance of applications. In this article we apply the lessons of the previous article to application codes and we quantify the performance benefit from thinking and coding in a "cache-aware" fashion.

Numerical techniques for PDEs

For a computational solution, the PDEs must be discretized and linearized, so that the problem becomes one of linear algebra: Solving a linear system with several millions of unknowns. These systems are too large to be solved by classical direct methods on any contemporary computer. Direct methods are not competitive on these very large systems, since they cannot efficiently exploit the sparsity of the system matrices. The methods of choice are all iterative. Instead of an elimination process they use repeated multiplications of the solution vector with the system matrix. Even if a significant, but limited number of iterations is required, the iterative methods are usually less costly than a direct method.

The most basic iterative solvers are the Jacobi and the Gauss-Seidel method. Even though they are uncompetitive with the more advanced methods, they are either useful as a model for more complex methods or as building blocks from which better methods are constructed.

In particular, this is true for multigrid methods, which are the fastest known solvers for many elliptic PDEs. For multigrid, a method like Gauss-Seidel is applied for several iterations. This is known as the smoother before coarser or finer grids are activated, where more smoothing steps are applied. Repeated execution of a simple method like Gauss-Seidel is responsible for most of the computational cost in a multigrid algorithm. It is commonly more than 80% of the computational cost of a simple multigrid solver.

Logically tensor product grid based problems

Many PDE problems today are still solved on uniform, tensor product, or logically uniform grids. The problems may be part of a domain decomposed grid with different mesh spacings in each of the domains. Many disciplines use grids like these in order to attain a high level of performance even though there may be more vertices in the global problem than by using an adaptively chosen, (quasi) unstructured grid.

If the grids are structured, it is natural to use a 2D or 3D array for storing the vector of unknowns. Besides, the system matrix can also be stored in a grid-oriented manner. In particular, if the PDE has constant coefficients, no matrix needs to be stored since all of the entries (corresponding to the same edge in the mesh) are identical. Consequently, if this is the case, only a few vectors (e.g., the solution, the right-hand side, and possibly the residual) have to be stored. In general, the memory needed is a small multiple of the size of the solution vector.

No matter how the data is stored, the fundamental memory access problem comes from how all iterative algorithms are designed. The core of these methods are repeated matrix-vector multiplications or simple variants. In some cases (e.g., relaxation methods like Gauss-Seidel) the matrix-vector multiplication is hidden because it is directly combined with an update of the solution vector.

Generally, the full data set for an approximate solution must be completely read from memory to be processed by the CPU and then be written back to memory. In the interesting cases, the vector plus the matrix are too large to be stored in any of the caches. In the process of accessing all vector and matrix elements, some of the older ones must be expelled from cache. Thus, when the sweep completes, and assuming an LRU (least recently used) strategy for cache replacement, the first vector and matrix elements cannot be in cache any more. Thus every sweep will have to start from scratch and load everything from main memory. This is required since there will be little to no data from the previous sweep that has not been displaced from the cache. Temporal locality can only be exploited when the whole data set is small enough to fit in the cache.

Consider a standard implementation of a 2D red-black Gauss-Seidel relaxation method based on a 5-point discretization of the Laplace operator, as shown in Figure 1. The matrix is not stored since we use a uniform mesh. The coefficients are typically kept in registers. The data which has to be loaded from main memory are two vectors, specifically the solution and the right-hand side vector. Note that this problem is particularly nasty to optimize since there are so few variables available.

```
double u(0:n,0:n), f(0:n,0:n)
do it = 1, It {
   do i = 1, n-1 {      // red nodes
     do j = 1+(i+1)%2, n-1, 2 {
       Relax( u(i,j) )
       } }
   do i = 1, n-1 {      // black nodes
     do j = 1+i%2, n-1 , 2 {
       Relax( u(i,j) )
       } }
   }
```

Figure 1: Standard implementation of red-black Gauss-Seidel.

The runtime behavior of the standard red-black Gauss-Seidel program on a Digital PWS 500au is summarized in Table 1. For the smallest grid size the floating point performance is relatively high compared to the peak performance of 1 Gflops. With increasing grid size the performance increases to approximately 450 Mflops to a 64×64 grid. Doubling the grid size to 128×128 reduces the performance to 200 Mflops. For even larger grids ($> 512 \times 512$) the performance further deteriorates to below 60 Mflops.

Grid size	Mflops	% of cycles used for					
		Exec	Cache	TLB	Branch	Depend	Nops
16	347.0	60.7	0.3	2.6	6.7	21.1	4.5
32	354.8	59.1	10.9	7.0	4.6	11.0	5.4
64	453.9	78.8	1.4	15.7	0.1	0.0	4.2
128	205.5	43.8	6.3	47.5	0.0	0.0	2.4
256	182.9	31.9	60.6	4.2	0.0	0.0	3.3
512	63.7	11.3	85.2	2.2	0.0	0.0	1.2
1024	58.8	10.5	85.9	2.4	0.0	0.0	1.1

Table 1: Runtime behavior of red-black Gauss-Seidel.

To detect why these performance drops occur, the program was profiled using the *Digital Continuous Profiling Infrastructure (DCPI)*. The result of the analysis is a quantification of processor cycles spent for execution (Exec), no-operations (Nops), and several different kinds of stalls (see Table 1). The stalls that are measured are data cache misses (Cache), data table lookaside buffer misses (TLB), branch mispredictions (Branch), and register dependencies (Depend).

For the smaller grid sizes the limiting factors are branch mispredictions and register dependencies. For the larger grid sizes the cache behavior of the algorithm has a large impact on the performance. For the largest grids data cache miss stalls account for more than 80% of all CPU cycles.

Since data cache misses are the principle factor for the disappointing performance of this code on larger problems, it is important to look at this effect. Table 2 shows the percentage of all array references that are satisfied by the corresponding levels of the memory hierarchy. To obtain the data we counted the total number of array references which occur in the relaxation method and measured the number of L1 data cache accesses as well as the number of cache misses for each level of the memory hierarchy, again using the *DCPI* tool.

The difference between the measured and the estimated number of L1 data cache accesses is shown in column "±". Small values can be interpreted as measurement errors. Higher values, however, indicate that some of the array references are not implemented as loads or stores, but as register accesses. The number of references which are satisfied by a particular level of the memory hierarchy is the difference between the number of accesses into

Grid size	Data Set Size (byte)	% of all accesses which are satisfied by				
		$\pm$	L1 Cache	L2 Cache	L3 Cache	Memory
32	17 K	4.5	63.6	32.0	0.0	0.0
64	66 K	0.5	75.7	23.6	0.2	0.0
128	260 K	-0.2	76.1	9.3	14.8	0.0
256	1 M	5.3	55.1	25.0	14.5	0.0
512	4 M	4.9	29.9	50.7	7.3	7.2
1024	16 M	5.1	27.8	50.0	9.9	7.2

Table 2: Memory access behavior of red-black Gauss-Seidel.

it (misses of the memory level above it) and the number of accesses which are not satisfied by it (misses for that particular memory level).

For the 32×32 and 64×64 grids the L1 and L2 caches clearly hold all the data. When the data no longer fits in the L2 cache some data must be fetched from the L3 cache. For the larger grids ($> 256\times256$), the data does not fit completely in the L3 cache.

The same principal governs processors with only one or two cache levels. Problems bigger than some threshold cause memory traffic to either the next level of cache or main memory.

The standard red-black Gauss-Seidel algorithm, Figure 1, repeatedly performs complete sweeps through the grid from bottom to top. One sweep updates all the red nodes and then another sweep updates all the black nodes. As we have already mentioned above, there is no temporal reuse between sweeps. However, a closer look reveals that there is some temporal locality within a sweep and that this is actually exploited. Assume that the grid is structured and that the cache is big enough to hold at least a certain number of grid lines. If a grid line is processed then data from neighboring grid lines are also accessed. Provided that the cache does not have associativity conflicts, it should contain the data for two of the three grid lines, since these two grid lines have already been used recently. Additionally, we can expect help from the caches in exploiting spatial locality. This is also obvious from Table 2, where even for the largest grids only 7.2% of the references go to main memory.

Improving the temporal locality of successive relaxation sweeps is not trivial since this requires the blocking of several iteration sweeps. In terms of linear algebra this is equivalent to computing several matrix-vector products simultaneously. This induces data dependencies, because in general the vector product Ax must be computed before $A(Ax)$ can be computed. Fortunately, PDE problems give rise to special matrices. There are several strategies how both Ax and $A(Ax)$ can be computed simultaneously.

Loop fusion and loop blocking are the standard techniques to improve the temporal locality for dense matrix algorithms. For the case of iterative algorithms for sparse matrices, this does not work so easily. Several iterations must be blocked, and this requires that the data dependencies be handled. The program must be transformed so that the original semantics are maintained. With the techniques described here, cache-aware algorithms will have bitwise identical results to the original algorithm, however, they will run faster. Modifications and variants of iterative algorithms that improve the cache performance are of interest as well, but modifications which alter the semantics are not the objective of this article.

We again consider red-black Gauss-Seidel relaxation. Assume that a 5-point stencil is placed over one of the black nodes. All of the red points that are required for relaxation are up to date provided the red node above it is up to date. Consequently, we can update the red nodes in any row i and the black nodes in row $i-1$ in pairs. This technique is a fusion technique: It fuses two consecutive sweeps through the grid, which update the red and black points separately, together to one sweep through the grid. Fusion techniques for red-black Gauss-Seidel were developed in the mid 1960s when the CDC 6600 became available. A comprehensive review of how to do this fusion for 5- and 9-point operators is contained in [2]. For codes see also `http://wwwbode.in.tum.de/Par/arch/cache`.

This 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 iteration to the next. If a 5-point stencil is placed over one of the red nodes in any line i, the node can be updated for the second time provided that all neighboring black nodes have been updated once. This is the case, as soon as the black node in line $i+1$ directly above the red node has been touched once. As described before, this black node can in turn be updated as soon as the red node in line $i+2$ directly above it has been updated for the first time. Consequently, we can update the red nodes in rows $i+2$ and i and the black nodes in rows $i+1$ and $i-1$ in pairs. This technique, a blocking technique, can be generalized to more than just two successive red-black Gauss-Seidel sweeps [1].

Both of the above techniques require that a certain number of rows fit entirely in the cache. The fusion technique assumes that the cache can hold at least four rows of the grid. The blocking technique assumes that at least $2m + 2$ rows of the grid fit in the cache, if m successive sweeps through the grid are performed together. This blocking and fusion can reduce the number of accesses to memory, but fail to utilize the higher levels of the memory hierarchy efficiently, especially the registers and the L1 cache. However, a high utilization of the registers and the L1 cache is crucial for very high performance of any algorithm.

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
Blocking (2)	21.1	29.1	43.6	4.4	1.8
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/p (4)	37.7	54.0	5.5	1.9	1.0

Table 3: Memory access behavior of different red-black Gauss-Seidel variants using a 1024×1024 grid.

Results for a problem with a 1024×1024 grid using the optimizations discussed are shown in Table 3. Clearly, the blocking strategies improve the effectiveness of the different cache levels. Fusing and blocking reduces the main memory accesses from 7.2% to only 1.2%. As expected, the working set is too large for the L1 cache and only L2 is big enough to hold the data. To further increase the performance we have developed a 2D blocking technique which is described in detail in [4]. It should utilize the L1 cache much better than the other strategies. If no array padding is used, however, conflict misses in all levels of the memory hierarchy prohibit any efficiency gain. This can be seen from the drastically increased number of memory accesses. In fact, this variant has by far the highest number of references that must be satisfied by main memory. Only when these conflict misses are eliminated by array padding, as shown in the row labeled "2D-Blocking/p", does the 2D blocking technique demonstrate its potential. Now 54% of all references are satisfied from L1 and an additional 37.7% are satisfied directly from the register set. Thus the efficiency of the L1 cache is now drastically improved and this program runs fastest of all those tested. This is also reflected in the performance gains obtained on the different codes, as shown in Table 4. The 2D blocking technique yields better performance, provided it is combined with array padding.

Implementation variant	Grid Size							
	16	32	64	128	256	512	1024	2048
Standard	347	355	454	206	183	64	59	56
Fusion	403	458	564	363	357	123	113	79
Blocking (2)	391	489	610	404	365	180	149	90
Blocking (3)	397	444	561	404	368	227	151	93
2D-Blocking (4)	337	331	361	315	299	46	87	57
2D-Blocking (4)/p	328	406	413	389	392	265	266	251

Table 4: Mflops for a 5-point Gauss-Seidel method with blocking and fusion on a Digital PWS 500au with a peak performance of 1 Gflops.

Problems on unstructured grids

Unstructured grids arise in many application areas for PDEs. Advocates of unstructured grids point to reduced grid generation time, geometric flexibility, and potential for adaptive refinement.

Unstructured grids result in linear systems which are sparse and unstructured. When assembled, such systems must be stored in a format such as compressed row or column format.A central feature of such formats is indirect addressing. This can lead to poor cache performance because of low spatial locality. Techniques used to optimize dense matrix operations cannot be applied to sparse matrix operations. A variety of techniques have been proposed to enhance the speed of sparse matrix operations, such as identifying repeating sparsity patterns in a preprocessing step.

We propose two strategies to enhance cache usage for the Gauss-Seidel method. Both have a preprocessing phase that blocks the underlying grid to find a matrix renumbering. The second phase in each strategy is a modified Gauss-Seidel method that takes advantage of the new ordering to increase cache performance. The modified Gauss-Seidel method returns the same bitwise answer as a standard Gauss-Seidel method based on the same mesh ordering.

Strategy 1: Fixed grid blocking for cache

The first step of the grid blocking strategy is to decompose the grid arising from the discretization into cache blocks. A cache block consists of nodes which form a connected set, i.e, between any two nodes i and j there is a path that is contained entirely within the set and has i and j as endpoints. Define the cache block boundary to be the subset of nodes of a cache block which are adjacent to nodes in another cache block. The distance of a node i is defined to be one more than the number of edges in a minimum length path between i and the cache block boundary.

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 we use is the METIS library.

After a cache block is identified, the next step is to identify how many Gauss-Seidel updates can be done on a particular unknown without violating data dependencies and without referencing data from another cache block. This problem is equivalent to identifying the distance of the associated grid node form the cache block boundary. If m updates are required, then any node with distance greater than m is said to have distance m.

Once the nodal distances have been found for all nodes and all cache blocks, the grid is renumbered. ¿From this renumbering, the matrix is reordered. Within a cache block, the nodes are partitioned into sub-blocks. Sub-block L_p is the set of all nodes of distance p from the cache block boundary. Assuming that m Gauss-Seidel smoothing steps will be performed, there are a total of m sub-blocks. The mesh nodes are renumbered beginning in the innermost sub-block L_m and working towards the cache block boundary. This ordering has the property that nodes which are closer to the block boundary have a higher number than those further from the boundary. The ordering is contiguous within blocks and sub-blocks.

We have bounded the maximum cost of finding distances for all nodes in a grid at approximately four fine-grid Gauss-Seidel updates if each node has one degree of freedom. If each node has two degrees of freedom, the bound is approximately one fine grid Gauss-Seidel update. With some additional information, such as which nodes are on the mesh boundary, the cost is even less. Experience shows that these bounds are pessimistic. The preprocessing time is negligible in actual examples.

In applying Gauss-Seidel each cache block is visited once, and as many updates are done as possible without referencing information from other cache blocks. In general, the nodes in sub-block L_i can be updated i times in this manner. All unknowns associated with sub-block L_m (the innermost sub-block) can be fully updated using information from within the cache block. For one Mbyte caches we have found experimentally that more than 50% of the unknowns in a cache block reside in the innermost sub-block, L_m. After each cache block has been visited once, the unknowns associated with sub-block L_m are fully updated, but the other unknowns have been updated fewer times. The modified Gauss-Seidel method backtracks through the cache blocks to finish updating these unknowns. The solution is the same, bitwise, as a standard Gauss-Seidel method sweeping through the entire grid with the same ordering.

Strategy 2: Bandwidth reduction scheme

An alternative to the grid-based blocking scheme is to reorder the underlying mesh by a bandwidth reduction algorithm. This strategy avoids the backtracking phase of the cache-aware Gauss-Seidel method described above. The size of the problem, however, may preclude using this method. The idea is motivated in [2]. We note that the idea of bandwidth reduction to enhance data locality is not new.

Define the bandwidth of a matrix $A = a(i,j)$ of order N to be $B = \max_{1 \leq i \leq N}\{\tau(i)\}$. Here $\tau(i) = \max\{j - i : a(i,j) \neq 0, j > i\}$. In the Gauss-Seidel method, the updated value of unknown i depends on the values of unknowns j, $j > i$ and $a(i,j) \neq 0$ from the previous iteration. Hence unknown i can have, at most, one more update than unknowns j, $j > i$ and $a(i,j) \neq 0$. In general, unknown i can be updated as soon as unknowns $i+1, \ldots, i+B$ have been updated.

The first step is to apply a bandwidth reduction scheme, such as Reverse Cuthill-McKee, to the adjacency matrix. The matrix and multigrid inter-grid transfer operators are then renumbered based on the new grid ordering.

Next, the unknowns are partitioned into subsets of contiguous unknowns.

The matrix rows are partitioned in the same way. The result is a partition $\{P_1, P_2, \ldots, P_N\}$, where P_k contains B contiguous rows. We require that the matrix, right-hand side, and unknowns associated with m consecutive subsets fit into cache. Here m is the desired number of updates. Otherwise this cache blocking method cannot be applied.

Finally, a modified Gauss-Seidel method is applied. The updating schedule is very similar to the fixed grid blocking strategy. As many updates as possible are done on $P_1, \ldots, P_m$. All unknowns in P_1 can be fully updated using data already present in cache. P_{m+1} is updated and subsets $P_m, \ldots, P_2$ are updated in order. At this point, P_2 is fully updated. Proceeding in this manner, m Gauss-Seidel updates are done. The unknowns in the last subsets must be treated in a slightly different manner. The benefit is increased reuse of data already present in cache while producing the same answer, bitwise, as a standard Gauss-Seidel method doing m updates by sweeping through the entire grid m times.

Numerical results for the unstructured grid techniques

All experiments were carried out on two different platforms. The first platform is an SGI Origin2000 with a R12000 ip32 processor, 128 Mbyte of memory and a 1 Mbyte 2-way set associative unified L2 cache. The second platform is one node of a HP SPP2200. One node is an HP 200 MHz PA-8200 chip with a 1 Mbyte direct-mapped data cache.

We solve a 2D linear elasticity problem on a domain in the shape of Austria. Forces are acting inwards from the northwest and the northeast. A portion of the northeast border is fixed with respect to x. Portions of the southeastern border and the south central border are fixed with respect to x and y. All other boundary conditions are homogeneous Neumann. Results are given in Table 5. More examples are in [3] and color pictures are linked in the Kentuky Full Caches (KFCs) page, `http://www.ccs.uky.edu/~douglas/ccd-kfcs.html`.

Acknowledgments

The work presented here — a truly international effort — was supported in part by the DFG Ru 422/7-1,2, NATO grant CRG 971574, and NSF grants DMS-9707040, ACR-9721388, and CCR-9902022.

smoother	HP SPP2200				SGI Origin2000			
	# relaxations							
	V(2,2)	V(3,3)	V(4,4)	V(5,5)	V(2,2)	V(3,3)	V(4,4)	V(5,5)
standard GS	1.27	1.60	1.92	2.21	3.31	4.27	5.46	6.26
cache-aware GS	0.80	0.92	1.04	1.18	2.00	2.33	2.83	3.09
Speedup	1.59	1.74	1.85	1.87	1.66	1.83	1.93	2.03

Table 5: V-cycle times for the fixed block Gauss-Seidel method on linear elasticity equation on Austria domain.

References

[1] C.C. Douglas, *Caching in with multigrid algorithms: Problems in two dimensions*, Paral. Alg. Appl., 9(1996), pp: 195–204.

[2] C.C. Douglas, J. Hu, M. Iskandarani, M. Kowarschik, U. Rüde and C.Weiss, *Maximizing cache memory usage for multigrid algorithms*, in Multiphase Flows and Transport in Porous Media: State of the Art, Springer-Verlag, Lecture Notes in Physics, eds; Z. Chen, R.E. Ewing and Z.-C. Shi, Berlin, 2000.

[3] C.C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Cache optimization for structured and unstructured grid multigrid*, ETNA, 10(2000), pp: 21–40.

[4] C. Weiss, W. Karl, M. Kowarschik, and U. Rüde, *Memory characteristics of iterative methods*, in Proceedings of the Supercomputing Conference, Portland, Oregon, 1999.

Craig C. Douglas douglas@ccs.uky.edu is a Professor at the University of Kentucky and affiliated with Yale University. Gundolf Haase ghaase@numa.uni-linz.ac.at is an Assistant Professor at Johannes Kepler Universität Linz and a Visiting Professor at the University of Kentucky. Jonathan Hu jhu@ccs.uky.edu is a Graduate Student at the University of Kentucky (and is moving to Sandia Laboratory shortly). Markus Kowarschik kowarschik@cs.fau.de and Ulrich Rüde ruede@cs.fau.de are a Research Assistant and a Professor, respectively, at the Universität Erlangen-Nürnberg. Christian Weiss weissc@in.tum.de is a Research Assistant at the Technische Universität München.