

# Portable Memory Hierarchy Techniques For PDE Solvers: Part II

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

The first part of this article, which appeared in the June issue of *SIAM News*, detailed the architecture and behavior of microprocessor caches. The impact of caches on the performance of applications was also made clear.

Here, in the second and concluding part of the article, we apply the lessons of the previous article to application codes, and we quantify the performance benefits that can be achieved by the user who thinks and codes in a “cache-aware” fashion.

## Numerical Techniques for PDEs

Computational solutions for PDEs involve discretization and linearization. The problem then 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 because they cannot efficiently exploit the sparsity of the system matrices.

## APPLICATIONS ON ADVANCED ARCHITECTURE COMPUTERS

*Greg Astfalk, Editor*

The methods of choice are all iterative. Instead of an elimination process, iterative methods use repeated multiplications of the solution vector with the system matrix. Even when a significant (but limited) number of iterations are required, an iterative method is usually less costly than a direct method.

The most basic iterative solvers are the Jacobi and the Gauss–Seidel methods. Although not competitive with the more advanced methods, they are useful either as models for more complex methods or as building blocks from which better methods can be constructed. The latter is true particularly for multigrid methods, which are the fastest known solvers for many elliptic PDEs. In multigrid, application of a method like Gauss–Seidel for several iterations—known as the smoother—

is followed by the activation of coarser or finer grids, with additional smoothing steps. Repeated execution of a simple method like Gauss–Seidel is responsible for most of the computational cost in a multigrid algorithm—commonly more than 80% for 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. A problem may be part of a domain-decomposed grid with different mesh spacings in each of the domains. Researchers in many disciplines use grids of this type to attain high levels of performance, even though the number of vertices in the global problem can be greater than it would be with an adaptively chosen, (quasi)unstructured grid.

With a structured grid, it is natural to use a two- or three-dimensional array for storing the vector of unknowns. The system matrix can also be stored in a grid-oriented manner. If the PDE has constant coefficients, no matrix needs to be stored—all the entries (corresponding to the same edge in the mesh) are identical. 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 is a consequence of the way all iterative algorithms are designed. The core of these methods consists of repeated matrix–vector multiplications or simple variants. In some cases (e.g., relaxation methods like Gauss–Seidel) the matrix–vector multiplication is hidden by being directly combined with an update of the solution vector.

Generally, the full data set for an approximate solution must be completely read from memory for use by the processor, and it must then be written back to memory. In the interesting cases, the vector and the matrix together are too large to be stored in any of the caches. As the vector and matrix elements are accessed, some of the older elements are expelled from the cache. With an LRU (least recently used) strategy for cache replacement, the first vector and matrix elements are no longer in cache when the sweep completes. Every sweep must start from scratch and load everything from main memory; all or almost all the data from the previous sweep will have been displaced from the cache. Temporal locality can be exploited only when the whole data set is small enough to fit into the cache.

Consider a standard implementation of a two-dimensional, red–black, Gauss–Seidel relaxation method based on a five-point discretization of the Laplace operator, as shown in Figure 1. The matrix is not stored, because we use a uniform mesh. The coefficients are typically kept in registers. The data that has to be loaded from main memory consists of the solution vector and the right-hand side vector. This problem is particularly difficult to optimize because so few variables are available.

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 good when compared with the peak performance of 1 Gflops. Increasing

the grid size improves the performance to approximately 450 Mflops on a  $64 \times 64$  grid. Increasing the grid size to  $128 \times 128$  reduces the performance to 200 Mflops. On even larger grids ( $>512 \times 512$ ), the performance deteriorates further, to below 60 Mflops.

To learn why these performance drops occur, we profiled the program, using the *Digital Continuous Profiling Infrastructure* (DCPI). DCPI quantifies the number of processor cycles for execution (Exec), no-operations (Nops), and stalls of several different kinds (see Table 1). Kinds of stalls 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 processor cycles.

Because data cache misses are the principal factor in the disappointing performance of this code on larger problems, we examined this effect more closely. Table 2 shows the percentages of all array references satisfied by the corresponding levels of the memory hierarchy. To obtain this data, we counted the total number of array references that 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 numbers of L1 data cache accesses is shown in the column labeled “±.” Small values can be interpreted as measurement errors. Higher values indicate that some of the array references are implemented not as loads or stores, but as register accesses. The number of references satisfied by a particular level of the memory hierarchy is the difference between the number of accesses into it (misses of the memory level above it) and the number of accesses not satisfied by it (misses for that particular memory level). For example, the number of references satisfied by the L2 data cache is the number of L1 data cache misses minus the number of L2 data cache misses.

Clearly, for the  $32 \times 32$  and  $64 \times 64$  grids, the L1 and L2 caches hold all the data. When the data no longer fits into the L2 cache, some data will have to be fetched from the L3 cache. For the larger grids ( $>256 \times 256$ ), the data does not fit completely into the L3 cache.

The same principle governs processors with only one or two cache levels. Problems above some threshold size cause memory traffic, either to the next level of cache or to main memory.

The standard red-black Gauss-Seidel algorithm (see Figure 1) repeatedly performs complete sweeps through the grid, from bottom to top. One sweep updates all the red nodes, and another then updates all the black nodes. As mentioned earlier, there is no temporal reuse between sweeps. A closer look reveals some temporal locality within a sweep, which 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. When a grid line is processed, data from neighboring grid lines is also accessed. Provided that the cache does not have associativity conflicts, it should contain the data for two of the three grid lines, because these two grid lines were used recently (i.e., encached). Additionally, we can expect help from the caches in exploiting spatial locality. This is also obvious from Table 2, which shows that 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 in that it requires the blocking of several iteration sweeps. In terms of linear algebra, this is equivalent to computing several matrix-vector products simultaneously. Data dependencies are induced in the process because, in general, the vector product  $Ax$  must be computed before  $A(Ax)$  can be computed. Fortunately, PDE problems give rise to “special” matrices, and several strategies are available for computing  $Ax$  and  $A(Ax)$  simultaneously.

Loop-fusion and loop-blocking are standard techniques for improving the temporal locality with dense matrix algorithms. For the case of iterative algorithms operating on sparse matrices, improving temporal locality is not easy. Several iterations must be blocked, which means that data dependencies will have to be handled. The program must be transformed in a way that maintains the original semantics. With the techniques described here, cache-aware algorithms will have results that are identical, bit-wise, to those of the original algorithm, but the modified algorithms will run faster. Modifications and variants of iterative algorithms that improve the

```
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.

Grid size	Mflops	Percentage 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.

Grid size	Data Set size (byte)	±	Percentage of all accesses satisfied by			
			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	60 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.

cache performance are of interest as well, but modifications that alter the semantics are not the objective of this article.

Consider again red–black Gauss–Seidel relaxation. Assume that a five-point stencil is placed over one of the black nodes. All the red points required for relaxation are up to date provided that the red node above the black node 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: Two consecutive sweeps through the grid, which would have updated the red and black points separately, are fused into one sweep through the grid. Fusion techniques for red–black Gauss–Seidel were developed in the mid-60s when the CDC 6600 became available. A comprehensive review of such fusion techniques for five- and nine-point operators can be found in [2]. For the actual codes, see <http://www.bode.in.tum.de/Par/arch/cache>.

This technique applies only to a 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 five-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 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 two successive red–black Gauss–Seidel sweeps [1].

Each of the techniques just described requires that a certain number of rows fit entirely into 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 into the cache, if  $m$  successive sweeps through the grid are performed together. Blocking and fusion can reduce the number of accesses to memory, but these techniques do not make efficient use of the higher levels of the memory hierarchy, especially the registers and the L1 cache. Efficient utilization of the registers and the L1 cache, however, is crucial for very good performance of any algorithm.

Results for a problem with a  $1024 \times 1024$  grid, with the optimizations discussed here, are shown in Table 3. Clearly, the blocking strategies improve the effectiveness of the different cache levels. Fusion and blocking reduce 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 performance, we developed a two-dimensional blocking technique, which is described in detail in [4]. Utilization of the L1 cache should be much better with this technique than with the other strategies. However, if no array padding is used, conflict misses in all levels of the memory hierarchy will preclude any efficiency gain. This can be seen in the table in the drastically increased number of memory accesses for this technique; 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 two-dimensional blocking technique demonstrate its potential; 54% of all references are now satisfied from L1, and an additional 37.7% are satisfied directly from the register set. The efficiency of the L1 cache is drastically improved, and this program runs the fastest of all those tested. This is also reflected in the performance gains obtained for the different codes, as shown in Table 4. The two-dimensional blocking technique yields better performance provided that it is combined with array padding.

### Problems on Unstructured Grids

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

Unstructured grids result in linear systems that are sparse and unstructured. When assembled, such systems must be stored in, for example, compressed row or column format. A central feature of such formats is the indirect addressing required, which 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; among them are a preprocessing step that identifies repeating sparsity patterns.

We propose two strategies for enhancing cache usage with the Gauss–Seidel method. Each has 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 an answer that is bit-wise the same as that of a standard Gauss–Seidel method based on the same mesh ordering.

Relaxation method	Percentage of all accesses satisfied by				
	±	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 with a  $1024 \times 1024$  grid.

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.** Performance (Mflops) for a five-point Gauss–Seidel method with blocking and fusion on a Digital PWS 500au with a peak performance of 1 Gflops.

### Strategy 1: Fixed Grid Blocking for Cache

The first step of the grid-blocking strategy is to decompose the grid that arises from the discretization into cache blocks. A cache block consists of nodes that form a connected set; that is, 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. The cache block boundary is defined as the subset of nodes in a cache block that are adjacent to nodes in another cache block. The distance of 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. 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 produce such decompositions. The package we use is the METIS library.

After a cache block has been identified, the next step is to determine the number of Gauss–Seidel updates that 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 from 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. This renumbering is used in the reordering of the matrix. Within a cache block, the nodes are partitioned into sub-blocks. Sub-block  $L_p$  is the set of all nodes at distance  $p$  from the cache block boundary. If  $m$  Gauss–Seidel smoothing steps will be performed, there will be a total of  $m$  sub-blocks. The renumbering of mesh nodes begins in the innermost sub-block  $L_m$  and works toward the cache block boundary. This ordering has the property that nodes closer to the block boundary have higher numbers than those farther 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 identification of the nodes that are on the mesh boundary, the cost is even less. Experience shows that these bounds are pessimistic; in actual examples, the preprocessing time is negligible.

In the application of Gauss–Seidel, each cache block is visited once and as many updates as possible are performed 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 with information from within the cache block. For 1-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, bit-wise, as that of 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 with a bandwidth-reduction algorithm. This strategy avoids the backtracking phase of the cache-aware Gauss–Seidel method just described. However, the size of the problem may preclude use of this method. Motivation for the idea can be found in [2]; we point out that the idea of bandwidth reduction to enhance data locality is not new.

We define the bandwidth of a matrix  $A = a(i,j)$  of order  $N$  to be  $B = \max_{1 \leq i \leq N} \{\tau(i)\}$ , where  $\tau(i) = \max\{j - i : a(i,j) \neq 0, j > i\}$ . 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  $i + 1, \dots, i + B$  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 intergrid 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  $\{P_1, P_2, \dots, P_m\}$ , where  $P_k$  contains  $B$  contiguous rows. We require that the matrix, the right-hand side, and the 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  $P_1, \dots, P_m$ . All unknowns in  $P_1$  can be fully updated with data already present in cache.  $P_{m+1}$  is updated, and subsets  $P_m, \dots, P_2$  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.

This scheme offers the benefit of increased reuse of data already present in cache while producing the same answer, bit-wise, 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 platforms: (1) an SGI Origin2000 with an R12000 ip32 processor, 128 Mbyte of memory, and a 1-Mbyte two-way set-associative unified L2 cache; and (2) one node of an HP SPP2200, consisting of an HP 200-MHz PA-8200 chip with a 1-Mbyte direct-mapped data cache.

We solved a two-dimensional linear elasticity problem on a domain in the shape of Austria. Forces act inward from the northwest and the northeast. A portion of the northeastern 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 can be found in [3], and color pictures can be viewed from the Kentucky 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.

### References

- [1] C.C. Douglas, *Caching in with multigrid algorithms: Problems in two dimensions*, Paral. Alg. Appl., 9 (1996), 195–204.
- [2] C.C. Douglas, J. Hu, M. Iskandarani, M. Kowarschik, U. Rde, 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, Z. Chen, R.E. Ewing and Z.-C. Shi, eds., Berlin, 2000.
- [3] C.C. Douglas, J. Hu, M. Kowarschik, U. Rde, and C. Weiss, *Cache optimization for structured and unstructured grid multigrid*, ETNA, 10 (2000), 21–40.
- [4] C. Weiss, W. Karl, M. Kowarschik, and U. Rde, *Memory characteristics of iterative methods*, in Proceedings of the Supercomputing Conference, Portland, Oregon, 1999.

Craig C. Douglas ([douglas@ccs.uky.edu](mailto:douglas@ccs.uky.edu)) is a professor at the University of Kentucky and is also affiliated with Yale University. Gundolf Haase ([ghaase@numa.uni-linz.ac.at](mailto:ghaase@numa.uni-linz.ac.at)) is an assistant professor at Johannes Kepler Universitt Linz and a visiting professor at the University of Kentucky. Jonathan Hu ([jhu@ccs.uky.edu](mailto:jhu@ccs.uky.edu)) is a graduate student at the University of Kentucky (and is moving to Sandia National Laboratory shortly). Markus Kowarschik ([kowarschik@cs.fau.de](mailto:kowarschik@cs.fau.de)) and Ulrich Rde ([ruede@cs.fau.de](mailto:ruede@cs.fau.de)) are a research assistant and a professor, respectively, at the Universitt Erlangen-Nrnberg. Christian Weiss ([weissc@in.tum.de](mailto:weissc@in.tum.de)) is a research assistant at the Technische Universitt Mnchen.

Smoother	HP SPP2200				SGI Origin2000			
	No. of 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 a linear elasticity problem on the “Austria” domain.