Cache–aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions

Christian Weiß[†], Markus Kowarschik[††], Ulrich Rüde[††], Wolfgang Karl[†]

[†] Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR–TUM)
Technische Universität München, Germany
{weissc,karlw}@in.tum.de

Universität Erlangen-Nürnberg, Germany
[††] Lehrstuhl für Systemsimulation (IMMD 10)
{kowarschik,ruede}@informatik.uni-erlangen.de

Abstract

Conventional implementations of iterative numerical algorithms, especially multigrid methods, merely reach a disappointing small percentage of the theoretically available CPU performance when applied to representative large problems. One of the most important reasons for this phenomenon is that the need for data locality due to poor main memory latency and limited bandwidth is entirely neglected by many developers designing numerical software. Only when most of the data to be accessed during the computation are found in the system cache (or in one of the caches if the machine architecture comprises a cache hierarchy) fast program execution can be expected. Otherwise, i.e. in case of a significant rate of cache misses, the processor must stay idle until the necessary operands are fetched from main memory, whose cycle time is in general extremely large compared to the time needed to execute a floating point instruction. In this paper, we describe program transformation techniques developed to improve the cache performance of two–dimensional multigrid algorithms. Although we merely consider the solution of Poisson's equation on the unit square using structured grids, our techniques provide valuable hints towards the efficient treatment of more general problems.

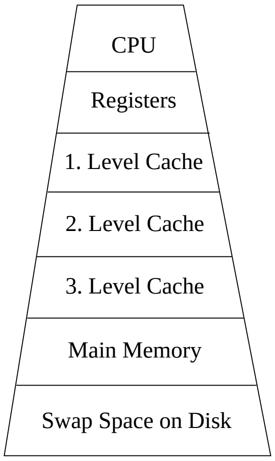
Key words. multigrid, cache memories, iterative methods, high performance computing, code optimization

AMS subject classifications. 68–04, 65F10

1 Introduction

Current semiconductor technology is capable of producing microprocessors operating with more than 500 MHz clock rate and a peak performance of beyond 1 GFlop (10^9 floating point operations) per second. The high performance of these chips is based on sophisticated hardware techniques like instruction pipelining, superscalar execution, out–of–order execution and dynamic branch prediction.

However, there is a serious bottleneck due to main memory access times. Current DRAM technology is not able to transfer data between processor and main memory as fast as necessary to avoid idle periods of the processor.



Capacity	Bandwidth	Latency
256 Bytes	24000 MB/s	2 ns
8 KBytes	16000 MB/s	2 ns
96 KBytes	8000 MB/s	6 ns
2 MBytes	888 MB/s	24 ns
1536 MBytes	1000 MB/s	112 ns

Figure 1: Memory hierarchy of the Digital Alpha PWS 500au A21164.

There is a rule of thumb saying that, in general, processor performance doubles every 18 months, whereas memory performance doubles every seven years [11]. Therefore, this problem is tending to become even worse in the near future, as for example prognosed in [15].

In order to mitigate these difficulties arising from slow main memory accesses, modern computers are equipped with a hierarchy of memories using one to three levels of comparatively fast SRAM caches, DRAM main memory and virtual memory, which is stored on external media such as hard disks or kept on other machines belonging to the same network. Cache memories contain copies of frequently used main memory blocks and provide low latencies whenever a cached data item is referred by the processor.

The use of these fast but rather small SRAM cache modules can be seen as a trade–off between price and performance: building the whole main memory using SRAM chips would be too expensive and, moreover, raise serious technical problems, especially concerning cooling techniques. Therefore, one will continue composing main memory out of DRAM modules, which are cheap, but unfortunately rather slow.

Most of our results presented in this paper are based on experiments using a Digital Workstation PWS 500au based on a 500 MHz Alpha 21164 processor. The theoretically available peak performance of this machine is about 1 GFlop/sec. Its memory hierarchy, the corresponding performance, bandwidth and capacity information are shown in Figure 1. The L1 cache and the L2 cache are integrated into the processor chip, whereas the large L3 cache is realized as a separate module, which is located on the board. This is the reason why fetching a data item from the L3 cache into the processor is significantly slower than fetching it from a higher level of the memory hierarchy.

It is evident that efficient program execution can only be expected if the algorithm respects the underlying memory architecture by exploiting the cache[1]: data in the cache must be reused before it is replaced. The effectiveness of data locality optimizations has been examined for various kinds of algorithms (see the examples in Section 3). However, research into the optimization of more advanced iterative methods has recently begun and is still far from being understood in detail [4, 9, 17].

[1]For reasons of simplicity we often use the singular form of the term *cache* to denote the whole hierarchy of caches that a machine may possess.

In this paper, we focus on a simple model problem and show that significant speedups can be achieved by introducing involved program restructuring techniques. These techniques are applied manually, since — generally speaking — optimizing compilers are not smart enough to perform such kinds of code transformations automatically. We consider the two–dimensional Poisson equation

$$\Delta u = f \text{ in } \Omega = (0,1)^2,$$

$$u = \phi \text{ on } \partial\Omega,$$

define a uniform grid and choose a second–order finite–difference discretization yielding the well–known 5–point stencil. We solve the problem numerically using multigrid V–cycles based on standard coarsening [6, 13]. We perform Gauss–Seidel iterations with a red–black ordering of the unknowns in order to smooth the approximate solutions on each level of the grid hierarchy.

We are aware that our model problem only represents a very special case. Our results should therefore be understood as indicators for what can be achieved by making efforts towards cache–aware implementations of these algorithms. In the case of more general situations, e.g. involving unstructured meshes, the situation is much more complicated. Nevertheless, significant speedups can also be gained by applying such acceleration techniques [10].

In Section 2 we consider the cache behavior of our relaxation procedure, which can be shown to be the computationally most intensive part of a multigrid cycle. We describe our data locality optimization techniques in Section 3 and present our results. In Section 4 we draw several final conclusions.

2 Cache behavior of the red–black Gauss–Seidel smoother

The efficiency of multigrid algorithms has been shown theoretically and in numerous practical applications. In terms of the number of arithmetic operations, multigrid methods are asymptotically optimal, i.e. the number of operations is only proportional to the number of unknowns. Profiling experiments confirm the smoother to be the computationally most expensive part of a multigrid algorithm. Therefore, improving the performance of this rather small code module can cause a significant speedup of the whole multigrid method.

The standard Gauss–Seidel relaxation algorithm for our model problem based on a red–black ordering of the unknowns is shown in Figure 2. Such a straightforward implementation of the red–black Gauss–Seidel algorithm iteratively passes once through the whole grid (for example from bottom to top) and updates all of the red points, then passes a second time through the entire grid and touches all of the black points. Usually, three to five red–black Gauss–Seidel sweeps (`noIter`) are performed before and after each grid transfer operation in the multigrid context.

The runtime behavior of our standard red–black Gauss–Seidel program on a DEC PWS 500au is summarized in Table 1. For the smallest grid size, the MFlop rate reaches less than one third of the theoretically available peak performance of 1 GFlop/sec. With the grids becoming larger the performance slightly increases to roughly 450 MFlops/sec. However, when reaching a grid size of 128×128, the performance dramatically drops down to less than 200 MFlops/sec, which is merely one fifth of the peak MFlop rate. For even larger grids ($\geq 512 \times 512$) the performance continues diminishing to poor 60 MFlops/sec.

To detect why those performance drops occur, we profiled our program using a tool named *DCPI* (*Digital Continuous Profiling Infrastructure* [2]). The result of this analysis was a breakdown of the total number of CPU

```
double u(0:n,0:n),f(0:n,0:n)

do it = 1 , noIter

  // red nodes:
  do i = 1 , n-1
    do j = 1+(i+1)%2 , n-1 , 2
      Relax( u(i,j) )
    enddo
  enddo

  // black nodes:
  do i = 1 , n-1
    do j = 1+i%2 , n-1 , 2
      Relax( u(i,j) )
    enddo
  enddo

enddo
```

Figure 2: Standard implementation of the red–black Gauss–Seidel method.

Grid Size	16	32	64	128	256	512	1024	2048
MFlops/sec	306.4	365.1	444.6	188.8	180.0	66.4	58.5	56.2

Table 1: Performance of Gauss–Seidel with red–black ordering on a DEC PWS 500au.

cycles in cycles spent for execution, nops and various kinds of stalls like for example data cache miss stalls and register dependency stalls. It turned out that, for the smaller grids, the limiting factors are dynamic branch misprediction and register dependency stalls. However, with growing grid sizes the cache behavior of the algorithm has an enormous impact on the runtime: for the three largest grids data cache miss stalls consume more than 80% of all the CPU cycles. This means that less than one fifth of the total program running time is spent for doing useful computations!

Since data cache misses are responsible for the poor performance of our straightforward red–black Gauss–Seidel algorithm on larger grids, it seems reasonable to take a closer look at its cache behavior. Therefore, we counted the number of array references in the algorithm and measured the number of cache misses for each level of the memory hierarchy, again using the *DCPI* profiling tool. In order to reduce the effects of unavoidable *compulsory cache misses*[2], we performed a large number of Gauss–Seidel iterations, i.e. we assigned a large value to the parameter `noIter`. Thus, we may assume that we measured almost only the occurrence of *capacity* and *conflict misses*[3]. Table 2 shows how many percent of all the array references are satisfied by each level of the memory hierarchy. The "±" column contains the differences of the percentages of manually counted and measured load operations. Small values in this column 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 analysis clearly shows that for the 32×32 and 64×64 grids, which both fit completely into the L2 cache of the memory hierarchy (see Figure 1, [8]), the algorithm can access all of the data from L1 and L2. But as soon as the data does no longer fit into L2, a high fraction of the data must be retrieved from L3. An analogous effect can be observed for problems which are too large to fit entirely into L3 (grid size $\geq 512 \times 512$). In this case, a certain percentage of the array elements has to be fetched from main memory. Thus, we conclude that the memory hierarchy cannot hold all of the repeatedly needed data close enough to the CPU as soon as the problem size increases.

Caches are designed to make use of the principle of *locality of data references* [14]. Iterative methods like Gauss–Seidel, however, perform successive global sweeps through typically very large data structures. As mentioned above, the standard red–black Gauss–Seidel method repeatedly performs one complete sweep through the grid updating all of the red nodes (from the bottom to the top of the grid) and then another entire sweep updating all of the black nodes (again passing from the bottom to the top). Assuming that the grid is too large to fit entirely into the cache, the data of the lower part of the grid is no longer in the cache after the update sweep for the red nodes, since they have been replaced by the data corresponding to the nodes in the upper part of the grid. Hence, the data must be reloaded from the slower main memory into the cache again. In this process newly accessed nodes replace the data corresponding to the upper part of the grid points in the cache and, as a consequence, they have to be reloaded from main memory themselves.

Although these kinds of iterative methods perform global passes through the whole data set, caches are nevertheless able to exploit at least some temporal and spatial locality [19]. For example, if we want to compute a new value for the black point located in the middle of Figure 3, we need the data from all of the red nodes in its neighborhood, the previous approximation of the black node itself and finally the corresponding value of the

[2]Whenever a data item is referred for the first time, it has to be fetched from main memory and brought into the cache. Hence, these types of cache misses are inevitable.

[3]*Capacity misses* occur when the cache is not large enough to hold all the data needed in the course of the computation. A *conflict miss* occurs when several data items are mapped to the same cache location (cache line) by the hardware and thus mutually force each other to be discarded.

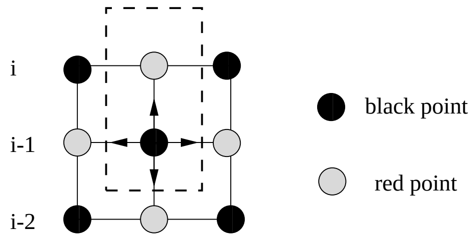


Figure 3: Data dependencies for Gauss–Seidel algorithm with red–black ordering.

right–hand side of the equation. The values of the red points in line $i-1$ and $i-2$ should still be in the cache because the black points in row $i-2$ have been updated recently. Moreover, the updated black node in line $i-1$ might be in the cache if the black node and the red node on its left side belong to the same cache line. The same argument applies to the red node in line i and the value of the right–hand side. Hence, the red node in line i, the value of the right–hand side and the black node in line $i-1$ have to be loaded from memory whenever the border of a cache line is crossed.

Looking at Table 2 one can derive two goals of locality optimization strategies for iterative methods. The first goal is to reduce the number of values which are fetched from the deepest level of the memory hierarchy into which the data set fits completely. In the case of a 512×512 grid this is the main memory for example. The second goal is to fetch a higher fraction of the data out of one of the higher levels of the hierarchy, including the CPU registers and the L1 cache.

The usual techniques to improve the cache utilization are compiler or hardware controlled prefetching [7], data access transformations [3], and data layout transformations [16]. Those techniques are well–known and implemented in many compilers (see for example [19]). But, in the case of iterative numerical methods, these automatic techniques do not even apply for simple program codes.

In the next section, we describe how the common data access transformations *loop fusion* and *loop blocking* as well as the data layout transformation *array padding* can be applied to our problem and what effects they have on the performance of our implementation. In this article, we will focus on the red–black Gauss–Seidel algorithm. However, we believe that similar data locality optimizations can be applied to other iterative methods.

3 Data Locality Optimizations

The effectiveness of data locality optimizations has already been demonstrated for matrix multiplication algorithms [5], for linear algebra algorithms for dense matrices [1], and for FFT algorithms [12], for instance. However, little work in that direction has been done for iterative techniques, and research into more advanced iterative methods such as multigrid has only just begun [4, 9, 17].

In the following we will describe how the well–known techniques *loop fusion* and *loop blocking* can be applied to the red–black Gauss–Seidel method. Afterwards, we will introduce a more involved transformation for this algorithm, which we call *windshield wiper technique* for reasons of clearness. The application of suchlike techniques to other kinds of smoothers like row or column Gauss–Seidel is part of our ongoing research. In the end of this section, we will focus on the problem of the elimination of cache conflict misses.

3.1 Fusion Technique

When implementing the standard red–black Gauss–Seidel algorithm, the usual practice is to perform one complete sweep through the grid from bottom to top updating all of the red nodes and then one sweep through the whole grid updating all of the black nodes. If a 5–point stencil is placed over one of the black nodes (see Figure 3), all of the red grid points that are needed for relaxing the black one are up to date, provided the red node above the black one is up to date, too. Consequently, we may update the red nodes in row i and the black nodes in row $i-1$ in pairs. It is evident that the red nodes in the first row of the grid and the black nodes in the last row require some special handling.

There are two ways to implement this idea. The first is to update each black node as soon as possible. This means that a red node in row i and the black node in row $i-1$ directly underneath the red one are touched in a pairwise manner. The second possibility is to first update a certain number of red nodes, which fit into the cache, and then compute the new values for all of the black nodes that are located underneath these red ones. For example, this can be done linewise.

Instead of doing an entire grid sweep touching all of the red points and then doing another full sweep updating all of the black points, we now perform a single sweep through the grid and touch the red and the black nodes alternatingly. Thus, the grid must be transferred from main memory to cache only once per update sweep instead of twice, as long as the rows needed for this pairwise update technique fit into the cache.

For the first implementation at least four rows of the grid must fit completely into the cache: three rows must fit into the cache to provide the data needed for updating the black points in the current row, and one additional grid row is necessary for updating the red points in the row above these three rows. For the second alternative it has to be guaranteed that the chosen number of grid rows plus three additional ones fit into the cache.

Analyzing the memory behavior of the first implementation variant for a 1024×1024 grid, we expect that this technique can halve the number of main memory accesses. The experimental results given in Table 3 indeed confirm our expectations: comparing the entries in the "Memory" column, one notices that the value belonging to the "Standard" row is twice as large as the entry in the "Fusion" row.

An additional effect which can improve the data locality of the first variant is that the compiler might keep the values corresponding to the current pair of nodes in processor registers, so that the update of the black node in row $i-1$ saves two load operations (see again Figure 3). Since 14 memory operations are required for the two update operations (six loads and one store operation for each update), roughly 15% of all the memory operations can be saved by a better register usage. In our case the Digital Fortran compiler in fact manages to reuse the data kept in registers. In the "$\pm$"–column of Table 3 we show how many of the memory operations are saved. One can observe that the number of register accesses has increased from 5.1% to 20.9% by introducing the fusion technique.

3.2 Blocking Technique

The technique described in the previous section optimizes the cache behavior of one single red–black Gauss–Seidel sweep. However, if several successive red–black Gauss–Seidel iterations have to be performed, the data in the cache is not reused from one iteration to the next, if the grid is too large to fit entirely into the cache.

A common data locality optimization technique, which is widely used in the field of array computations, is *loop blocking* (also called *loop tiling*). The loop is transformed in such a way, that the computations are performed successively on *blocks* (or *tiles*) of the array, rather than on the whole array at once. The size of the blocks is

Grid Size	Data Set Size	% of all accesses which are satisfied by				
		±	L1 Cache	L2 Cache	L3 Cache	Memory
32	17 KByte	4.5	63.6	32.0	0.0	0.0
64	66 KByte	0.5	75.7	23.6	0.2	0.0
128	260 KByte	-0.2	76.1	9.3	14.8	0.0
256	1 MByte	5.3	55.1	25.0	14.5	0.0
512	4 MByte	4.9	29.9	50.7	7.3	7.2
1024	16 MByte	5.1	27.8	50.0	9.9	7.2
2048	64 MByte	4.5	30.3	45.0	13.0	7.2

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

Relaxation Method	% of all accesses which are 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
Wiper (4)	36.7	25.1	6.7	10.6	20.9
Wiper/p (4)	37.7	54.0	5.5	1.9	1.0

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

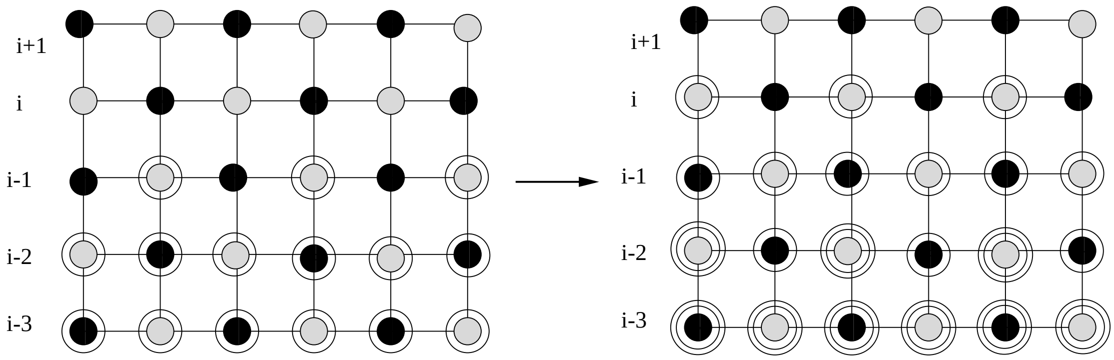


Figure 4: Example of blocking two red–black Gauss–Seidel sweeps. The circles around a node denote how often it has already been touched.

adapted to the size of the cache, and the data within a block is reused as often as possible before the process of computation moves on to the next block of the array.

Unfortunately, the loop blocking technique is not directly applicable to iterative methods because of data dependencies between neighboring nodes of the grid. In the case of the standard red–black Gauss–Seidel method based on a 5–point discretization of the Laplacian, a node of the grid cannot be touched for the second time before all of its four direct neighbors have been updated for the first time. Hence, if the nodes belonging to a certain block are to be updated several times, the block must necessarily shrink by two rows (one row with red points and one with black points) after each iteration. In order to keep a constant sized block, it must slide through the grid.

After a red node in row i has been updated for the first time (see left side of Figure 4) the black node in row $i-1$ directly underneath the red one can also be updated for the first time. As a consequence, the red node in row $i-2$ and the black node in row $i-3$ belonging to the same column of the grid can be updated for the second time, and so on. Eventually, the red node in row $i-m*2+2$ and the black node in row $i-m*2+1$ will be updated for the mth time. This may be done for all of the red nodes in row i (see right side of Figure 4). In the next step, the whole block is moved up to row $i+1$. Of course, some border handling is required in the lower and in the upper part of the grid.

Instead of successively passing m times through the whole grid and updating each node once per sweep, we now move only once through the grid and touch each node m times. Hence, in the case of a size of 1024×1024 the grid is transferred from main memory to the cache once instead of m times. However, this is only correct if $m*2+2$ rows of the grid fit into the cache. Therefore, the percentage of main memory accesses needed for red–black Gauss–Seidel using the blocking technique should be roughly the percentage of main memory accesses needed for the fusion technique divided by the number m of blocked iterations. The "Blocked (2)" row and the "Blocked (3)" row of Table 3 show that the blocking technique can indeed halve resp. third the number of memory accesses needed for the red–black Gauss–Seidel method.

An overview of the floating point performance of standard, fused and blocked red–black Gauss–Seidel is presented in Figure 5. Both optimization techniques can improve the performance for all grid sizes. For the small grids, which fit entirely into the L2 cache, the speedup is marginal. Nevertheless, the rate of 600 MFlops/sec for a 64×64 grid is quite impressive. The greatest speedup is achieved for the grids which do not fit into the L2 cache and for the 512×512 grid which is the first grid that does not even fit into L3. For the 512×512 grid we obtain a speedup

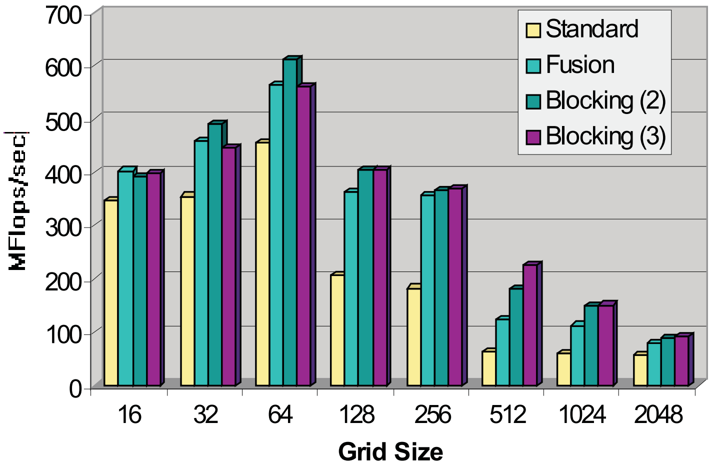


Figure 5: MFlops/sec for different red–black Gauss–Seidel implementations.

factor of 3.6. However, none of the techniques can significantly increase the performance of the algorithm for the very large grids. This is due to the fact, that all techniques assume a certain number of grid rows to fit into the cache. For the 2048×2048 grid, however, less than six rows fit into L2. Therefore, the blocking technique is not working well anymore. Besides, the goal of a high L1 cache utilization is not yet reached.

3.3 Windshield Wiper Technique

Both techniques described in the previous sections reduce the number of references to the highest level of the memory hierarchy into which the whole grid fits. In the case of a 1024×1024 grid on a DEC Alpha PWS 500au this is the main memory. Thus, both techniques mainly optimize for L3. However, they fail to utilize the higher levels of the memory hierarchy efficiently — particularly the registers and the L1 cache. We believe that a high reuse of the contents of the registers and L1 is crucial for the performance of iterative methods. Our windshield wiper technique straightly addresses this issue.

The main idea behind the windshield wiper technique is to use a special two-dimensional blocking strategy. A tiny two-dimensional block is moved through the grid updating all the nodes which it covers. The high utilization of the registers and the L1 cache is achieved by overlapping these blocks as much as possible, still respecting all data dependencies. The principle of the technique will be described in the following by an example which updates all the nodes in the grid twice during one global sweep.

For exposition, consider the situation shown in Figure 6. Assume that a correct initial situation has already been set up. The red and the black points on the lower two diagonals of the left parallelogram have already been updated for the first time, whereas its upper two diagonals are not yet touched. Note that — due to the 5–point discretization of the Laplacian — the nodes belonging to the same diagonal may be updated independently from each other. This reduces the number of data dependencies the compiler has to consider. Firstly, the upper two diagonals of the left parallelogram are processed: as soon as the red points in the uppermost diagonal are all updated for the first time, the black points in the diagonal below may also be updated for the first time.

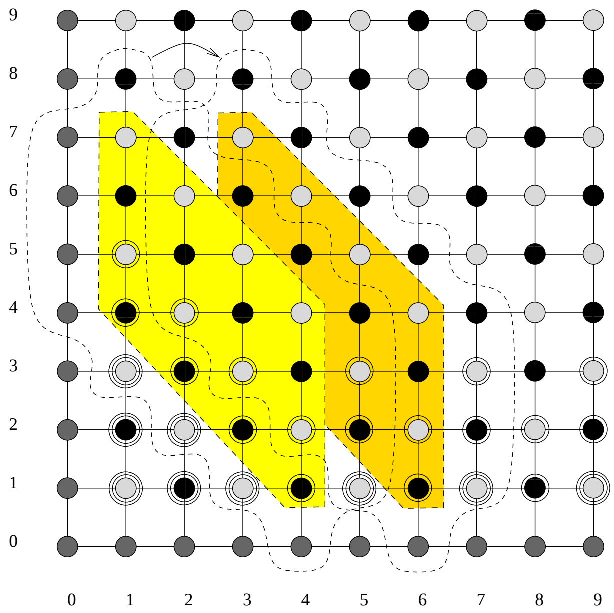


Figure 6: Windshield wiper technique for red–black Gauss–Seidel.

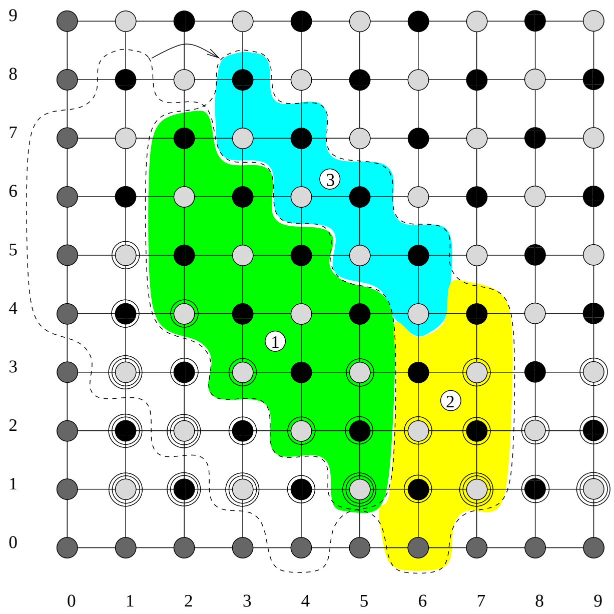


Figure 7: Data region classification for the windshield wiper technique.

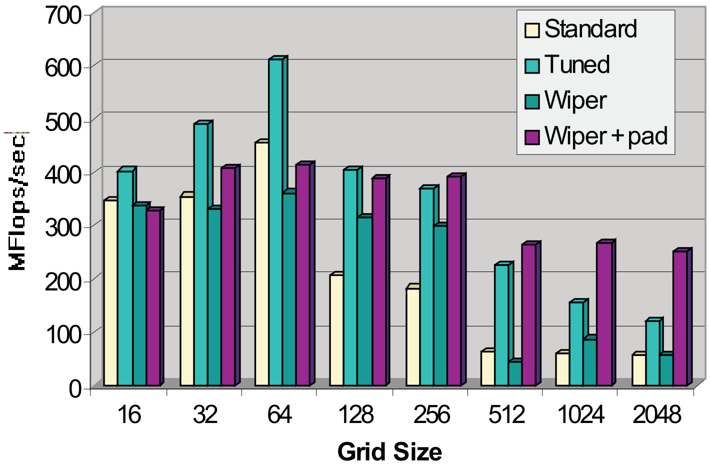


Figure 8: MFlops/sec for windshield wiper red-black Gauss-Seidel.

Consequently, the red nodes and the black nodes on the lower two diagonals may be updated for the second time.

Now, the situation in the right parallelogram is as has been before in the left one: the nodes on the two lower diagonals are already updated once, the two upper diagonals are still untouched. Consequently, we switch to the right parallelogram and start updating the grid nodes which it covers analogously. In that fashion, the parallelogram is moved through the grid until it reaches the right boundary. There, we do some boundary handling and move the parallelogram four rows up. After that, some boundary handling on the left side is necessary, before we begin anew with moving the block from left to right. By this means, we move through the whole grid — similar to the motion of a windshield wiper — updating all the nodes twice.

The grid points needed for updating all the nodes belonging to one parallelogram are characterized by the outermost dotted line in Figure 6. The nodes within the overlapping region of two adjacent dotted lines (area 1) are those which are directly reused when switching from one parallelogram to the next. The size of this overlap region depends on the number of blocked update sweeps (in the sense of 3.2). In standard multigrid algorithms, only few (typically e.g. four) smoothing steps are applied on each level. Thus, the size of that data should be small enough to fit at least into the L1 cache. In our example, 128 Bytes are sufficient[4]. Area 2 can be reused from a previous lateral move through the grid, as long as enough grid rows can be kept in one of the levels of the memory hierarchy. Typically, the data for area 2 is stored in one of the deeper (and slower) levels of the memory hierarchy. The rest of the data (area 3), however, must be fetched from main memory if the whole grid does not fit entirely into one of the caches.

The performance of the windshield wiper algorithm with four blocked iterations is shown in Figure 8. It is compared to the standard implementation and to the best implementation achieved by applying the fusion and the blocking technique (abbreviated by the "Tuned" predicate). Unfortunately, the approach does not work very well. For the small grids and for the larger grids the floating point performance is equal to or even worse than the

[4]There are 16 nodes (unknowns) within the overlap region, and a double precision floating point number occupies 8 Bytes of memory on the DEC machine.

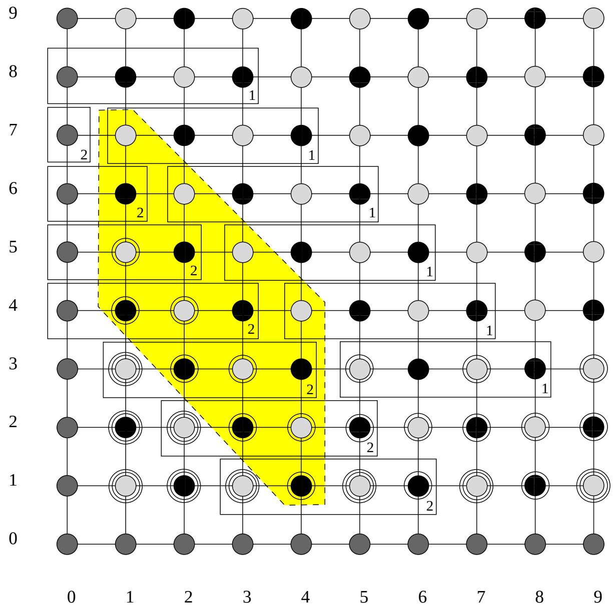


Figure 9: Alpha 21164 L1 cache mapping for a 1024 × 1024 grid.

performance of the standard implementation!

An analysis of the memory behavior with *DCPI* — summarized in the "Wiper (4)" row of Table 3 — revealed that, although the utilization of the registers has improved, the L2 cache utilization has decreased dramatically. As a consequence, more than 20% of all array references cannot be cached and thus cause main memory accesses. The reason for this is a very high number of cache conflict misses. Applying the cache visualization tool CVT [18] one can observe that throughout the whole runtime of the program only four (out of 256!) cache lines of the direct mapped L1 cache are used simultaneously. Surprisingly, even the L2 cache, which is 3–way set associative in the case of the Digital Alpha 21164, is not able to resolve the conflict misses[5].

The mapping of the nodes covered by a parallelogram onto the cache lines of the DEC Alpha 21164 L1 cache is presented in Figure 9 for a grid size of 1024 × 1024. All the elements of the uppermost diagonal are mapped to the same cache line (cache line #1). Hence, the data of the uppermost diagonal is not reused during the update of the four diagonal elements below. Furthermore, the data needed for the update of a single node is conflicting in the L1 cache. For example, when updating the red node (1,7), the nodes (0,7), (1,8), (2,7), (1,6), and (1,7) are needed. Thus, references to the nodes (0,7) and (1,6), (1,8) and (2,7), as well as (1,8) and (1,7) imply conflict misses in L1.

One of the common techniques to reduce the effect of conflict misses is *array padding*. Hence, we apply this technique to the two two–dimensional arrays (unknown vector and right–hand side vector) of our windshield wiper Gauss–Seidel algorithm: we introduce additional array elements that are never referred during the process of computation. As the darker shaded columns in Figure 8 illustrate, we obtain a remarkable speedup. Especially for the large grids the technique achieves a speedup factor of 4.6 compared to our standard implementation, yielding a floating point performance of about 267 MFlops/sec for a grid size of 1024 × 1024. The "Wiper/p (4)" Row of Table 3 shows indeed that the padded windshield wiper technique utilizes the registers and the L1 cache much better than all the other techniques. With this code, more than 90% of all accesses are satisfied by the L1 cache

[5] In an n–way set associative cache, there are n cache locations (cache lines) where a data item may be stored.

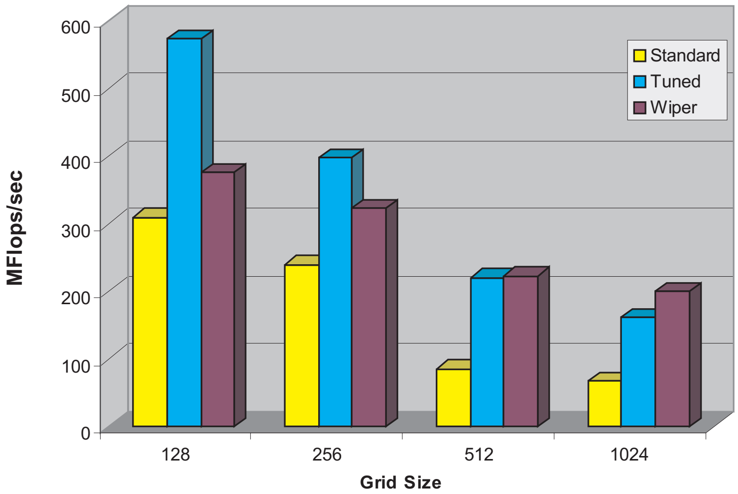


Figure 10: MFlops/sec for a multigrid method based on standard V–cycles for solving Poisson's equation on a DEC PWS 500au.

or the processor registers.

Integrating our fast relaxation methods into a multigrid code that iteratively performs V–cycles to solve Poisson's equation on the unit square we obtain the results presented in Figure 10. The most impressive observation is that for a grid size of 1024×1024 the code runs roughly four times faster than before by applying the padded windshield wiper transformation to the relaxation procedure.

Table 4 affirms that our efforts are not restricted to the DEC architecture at all. We also obtain remarkable speedups for the red–black Gauss–Seidel relaxation on a SUN Ultra 60 and on an SGI Origin 2000 node by applying our program transformation techniques. For each machine and each grid size, we compare the MFlop rate of a standard implementation (according to Figure 2) with the best one obtained by making use of the data locality optimizations we described above. Of course, the speedups depend on the memory architecture of the machine and on the ratio of memory speed to floating point peak performance. For example, the DEC memory architecture comprises three cache levels, whereas the memory subsystems of the SUN and the SGI machines only use two levels of cache memories. Therefore, the speedup values in the "DEC" part of Table 4 show two significant jumps, while the speedup values in the "SUN" part and the "SGI" part merely show one noticeable performance jump. Generally speaking, the gap between floating point peak performance and memory speed is larger for the DEC machine than for the other two architectures. This is the reason for the higher speedups that can be achieved on the DEC workstation by cache–aware programming.

3.4 Problems in 3D

In general, when working with three–dimensional arrays the problem of the occurrence of conflict misses tends to become more serious than in the two–dimensional case.

Consider a square grid that is implemented as a two–dimensional array of double precision values. Thus, two array elements which are adjacent in the leading array dimension are also adjacent in virtual address space,

Grid Size	SUN			SGI			DEC		
	MFlops/sec		Speedup	MFlops/sec		Speedup	MFlops/sec		Speedup
16	132.17	173.16	1.3	187.02	252.33	1.3	306.40	385.51	1.3
32	152.22	185.24	1.2	207.26	287.30	1.4	365.10	507.65	1.4
64	99.28	169.73	1.7	146.96	260.58	1.8	444.64	584.52	1.3
128	102.43	154.09	1.5	151.92	261.62	1.7	188.76	440.43	2.3
256	102.05	143.20	1.4	173.57	250.32	1.4	179.96	426.15	2.4
512	51.62	113.60	2.2	141.55	195.32	1.4	66.35	263.76	4.0
1024	45.60	113.40	2.5	66.90	163.21	2.4	58.50	267.91	4.6
2048	42.23	112.68	2.7	66.79	150.50	2.3	56.22	250.92	4.5

Table 4: Speedups for red–black Gauss–Seidel on a SUN Ultra 60 (296 MHz), an SGI Origin 2000 Node (195 MHz) and a DEC PWS 500au.

whereas the distance in the virtual address space between two array elements that are neighbored in the trailing array dimension is determined by the number of points per grid row (= number of elements per array dimension).

For a cubic grid that is implemented as a three–dimensional array, two array elements which are neighbors in the trailing array dimension are quite far away from each other in virtual address space and therefore more likely to cause cache conflicts.

The following example is to illustrate this problem. Consider a cubic grid containing 64^3 points that is realized as a three–dimensional array of double precision values, each of which is assumed to occupy 8 Bytes of memory. Let x be the leading and z be the trailing array dimension. Moreover, our machine is assumed to have an 8 KByte direct–mapped cache memory (e.g. the Digital Alpha 21164 processor again). Then, array elements that are adjacent in x–direction are also adjacent in virtual address space; array elements that are adjacent in y–direction are $64 * 8$ Bytes away from each other. However, the worst effect is that the distance in address space of z–neighbored array elements is 2^{15} Bytes, which is exactly four times the size of the cache memory! This means, that every pair of array elements that are adjacent in direction z map to the same cache location and therefore mutually throw each other out of the cache. Of course, this effect significantly decreases the performance of stencil operations that are performed within the grid.

There are several approaches to avoid this effect. It might be possible to use sophisticated padding strategies that are based on individually placing the grid rows in main memory. This seems to be a significant implementational overhead. Alternatively, there is the possibility to use non–standard (wrapped) data types to store the grid, that are different from three–dimensional arrays.

4 Conclusions

In this paper we have outlined cache optimization techniques that are useful for iterative methods applied to systems of linear equations that arise in the context of numerical PDE solution. Our focus was explicitly on multigrid methods, since these are the fastest known algorithms for these kinds of problems in terms of floating point operations per unknown. However, our coding techniques can be used in other iterative solvers, since the computational kernels are similar in many iterative methods.

Repeated Gauss–Seidel relaxations, as they are used as smoothers in multigrid algorithms, can be restructured to obtain a good reuse out of quite small working sets. Our windshield wiper technique is optimized for caches as small as the 8 KByte L1 cache, e.g. found in the DEC Alpha 21164 chip and the primary working set size is independent of the grid size. This is accomplished by carefully designed reorganization of the processing order, combined with array padding to avoid cache associativity conflicts.

The effectiveness of this technique is demonstrated by extensive profiling. More than 250 MFlops/sec and thus 25% of the machine peak performance can be maintained independently of the grid size. This applies to a code that implements four successive relaxation steps in just a single pass through the grid. We consider this to be a quite good result, since the sparse matrix operations in an iterative method have much less potential for data reuse than full matrix algorithms.

The multigrid method based on the optimized relaxation performs a full multigrid algorithm [13] at approximately 0.5 microseconds per unknown on the DEC Alpha 21164, and can thus solve a Poisson problem with a million unknowns up to truncation error accuracy in about 0.5 seconds. Compared to a non cache–optimized version of the code, this is an improvement by a factor 4. Though the code has been developed specifically for the Alpha 21164 cache architecture, similar performance results and improvements are obtained for other high performance architectures, as for example SUN Ultra and SGI Origin machines.

The optimization of three–dimensional grid codes turns out to be significantly more complicated since any straightforward blocking strategy seems to dramatically increase the number of conflict misses. Other than in two–dimensional case, these associativity conflicts cannot be avoided by array padding. Alternative memory layouts are currently being investigated and the extension of our techniques to more general problems, involving more general problem types and grid structures, is also planned for the near future.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

[2] 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, October 1997.

[3] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM computing surveys*, 26(4):345ff, December 1994.

[4] F. Bassetti, K. Davis, and D. Quinlan. Temporal Locality Optimizations for Stencil Operations within Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures. In *Proceedings of the PDCS'98 Conference*, Las Vegas, Nevada, July 1998.

[5] 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.

[6] A. Brandt. Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics. *GMD Studien*, 85, 1984.

[7] T-F. Chen and J-L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the International Symposium on Computer Architecture*, pages 223–232, 1994.

[8] Digital Equipment Corporation, Maynard, Massachusetts. *Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual*, 1997. Order Number: EC-QP99B-TE.

[9] C. C. Douglas. Caching in With Multigrid Algorithms: Problems in Two Dimensions. *Parallel Algorithms and Applications*, 9:195–204, 1996.

[10] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transactions on Numerical Analysis*, 9, 2000.

[11] K. Dowd and C. Severance. *High Performance Computing*. O'Reilly, Sebastopol, 2nd edition, 1998.

[12] M. Frigo and S. G. Johnson. The Fastest Fourier Transform in the West. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, September 1997.

[13] W. Hackbusch. *Multigrid Methods and Applications*. Springer Verlag, Berlin, 1985.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher, Inc., 2nd edition, 1996.

[15] D. Keyes, D.K. Kaushik, and B.F. Smith. Prospects for CFD on Petaflops Systems. Technical Report 97–73, ICASE, NASA Langley Research Center, December 1997.

[16] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, Montreal, Canada, July 1998.

[17] 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 The Eighth Biennial Computational Techniques and Applications Conference*, Adelaide, Australia, September 1997.

[18] E. van der Deijl, O. Temam, E. Granston, and G. Kanbier. The Cache Visualization Tool. *IEEE Computer*, July 1997.

[19] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.