Computational Techniques and Applications: CTAC97
World Scientific

Data Local Iterative Methods For The Ef cient Solution of Partial Differential Equations

Linda Stals, Ulrich Rüde
Institut für Mathematik, Universität Augsburg, D-86159 Augsburg, Germany

Christian Weiß, Hermann Hellwagner
Institut für Informatik, Technische Universität München, D-80290 München, Germany

1. Introduction

The continuing evolution of computer technology results in an increasing gap between CPU speed and memory performance. Current standard microprocessors operate at a peak performance of up to and beyond 1 GFlops (10^9 floating point operations per second). At the same time the typical peak main memory bandwidth is just 1 GByte/sec. For supporting (daxpy-type) vector operations at peak CPU speed, 24 times as much would be necessary. Thus the main bottleneck in these architectures is the memory performance.

Such machines use a *memory hierarchy* consisting of (possibly several layers of) caches. A cache is a small but fast memory that is either integrated into the CPU chip or built from high speed SRAM chips. An access to data not residing in the cache has to pay the extra cost of copying the information from the main memory to the cache. Caches thus help to speed up programs only when cached data is being reused. For programs that do not re-use data, the speed of processing is limited by the poor latency and bandwidth values of the main memory. For many compute intensive applications it has therefore become important to write cache-aware programs. For more information on technical issues, see [8, 3].

Multigrid is one of the most attractive algorithms for the solution of large sparse systems of equations that arise in the solution of partial differential equations (PDE). 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*, that is the number of operations is only proportional to the number of unknowns, see for example [6]. In standard problems, like scalar elliptic PDE in two or three space dimensions, multigrid can compute a solution within 100 floating point operations per unknown, and in special situations like Poisson's equation even in below 30 operations per unknown. On a GFlops machine, we would thus expect to solve a system of one million unknowns in a tenth of a second or below. Unfortunately this figure is much too optimistic, since all multigrid codes known to the authors fail to reach the peak performance on cache based machines by a large margin.

In this paper we shall therefore show how non-standard cache-aware implementations of the multigrid method improve the out-of-cache performance significantly. After a brief introduction to multigrid methods, we will discuss cache-oriented programming techniques and their effect on multigrid performance. Similar techniques are described in [4].

2. The Multigrid Method for the Solution of Elliptic Partial Differential Equations

2.1. Problem description In this article, we consider the finite difference approximation of a model elliptic partial differential equation defined on the square domain $\mathcal{M} = [0,1] \times [0,1]$ with Dirichlet boundary conditions. The finite difference method uses a uniform discrete grid with mesh-size $h_k = 2^{-k}$. The multigrid method will use a sequence of grids with increasingly finer mesh-size $k = 1, \ldots, n$.

We assume that the finite difference stencil is constant on each grid level and may be represented in compact 9-point form,

$$\begin{bmatrix} s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 \\ s_7 & s_8 & s_9 \end{bmatrix}.$$

This restricts the discussion in the present paper to linear PDE with constant coefficients, though of course many of the basic ideas can be carried over to more general situations for example those involving variable coefficients.

In most of this paper we will use the compact 9-point *Mehrstellen*-stencil of the form

$$\frac{1}{6}\begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}.$$

In our context, this has two interesting features. If additionally to using the *Mehrstellen*-operator, the right hand side is preprocessed by applying the averaging operator

$$\bar{R}_k = \frac{1}{6}\begin{bmatrix} 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 4 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 \end{bmatrix},$$

then an approximation accuracy of $O(h^4)$ is obtained, where h is the mesh-size of the grid. This requires the solution to be smooth enough, see e.g. Hackbusch [6]. Clearly, the 9-point discretisation is more expensive in terms of operations required. For smooth enough solutions, however, its better convergence order leads to more efficient algorithms. Additionally, our results indicate that the compact 9-point stencil results in a higher Mflop rate than the 5-point stencil. This effect mostly compensates the higher work count, making the method additionally attractive in our context.

Multigrid methods use multiple layers of grids with each grid layer removing different frequency components of the error. For detailed information about these algorithms, in particular their mathematical properties, we refer to the standard literature [1, 2, 6, 7].

We use a standard multigrid V-cycle, as defined by Brandt [2] or Hackbusch [6], following the standard construction using bilinear interpolation

$$I_{k-1}^{k} = \frac{1}{4}\left[\begin{array}{ccc} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{array}\right],$$

and set the restriction operator to be the transpose of the interpolation operator $R_k^{k-1} = (I_{k-1}^k)^T$.

Additionally, the V-cycle will be used in a *nested iteration* in the form of a so-called *FMV-scheme*, see Brandt [2]. The interpolation operator $\widetilde{I}_{k-1}^k$ used in the FMV-scheme has to be of high order to make sure that we do not introduce high frequency errors to the fine grid, see [1, 2]. Therefore we use fourth order polynomial interpolation to ensure that we do not loose the $O(h^4)$ accuracy offered by the compact 9-point stencil.

3. Environment and Data Structures

The test examples were run on an SGI Power Indigo based on the R8000 processor running at 75 Mhz. This machine has a peak CPU performance of 300 Mflops. It is equipped with a first level data cache of 16 Kbytes that is only used for integer data. Floating point data is directly accessed from 2 MB of second level external cache. We have used the CC compiler using options providing machine specific and application specific code optimisation[1].

The multigrid levels are stored in an array of grids. The arrays are allocated as single blocks of memory. Independent pointers are introduced for access the consecutive rows to avoid index expressions of the form `[i*number_columns+j]`.

4. Red/Black Gauss Seidel

4.1. In-Cache Optimisations. Figure 1 shows the Mflops rate for different implementations of the red/black algorithm. The line labelled `red black(1)` is the result of a standard implementation of the red/black Gauss Seidel method that is used as the starting point for our investigation. According to our choice of C/C++ as implementation language, we assume that the arrays are stored row-wise, so that the column-indices (`j`) are used in the inner loop. The grid is assumed to have dimensions `[0, grid size]` $\times$ `[0,grid size]` with the boundary lines holding the given Dirichlet data.

[1]Compiler flags: `-Ofast=ip26 -OPT:alias=restrict`. The first flag tells the compiler to optimise the code for the specific CPU chip (rather than generating generic object code), the second that the pointers will not alias, thus enabling more aggressive use of CPU-internal parallelism.

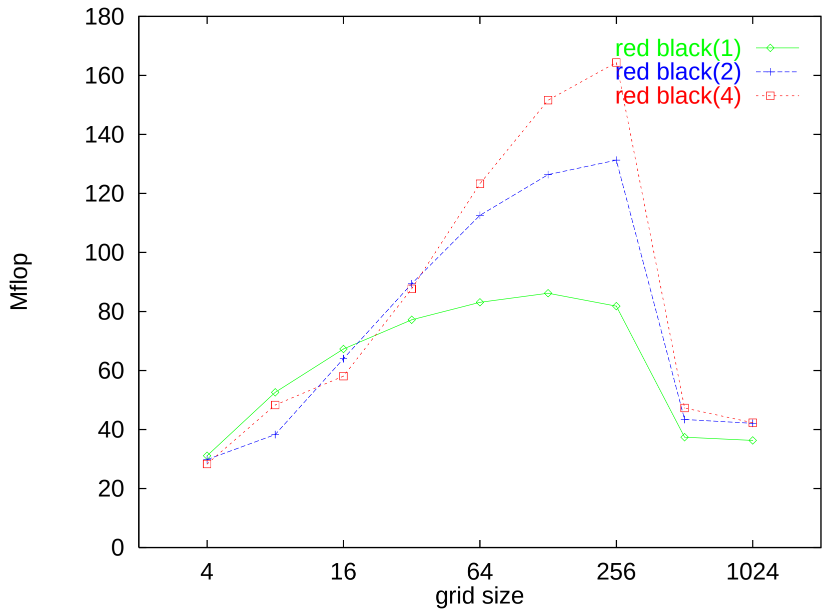

FIGURE 1. Mflops rate for different unrolled versions of red/black Gauss Seidel .

Standard Red/Black Gauss Seidel Algorithm:
update the red points
`for (i = 1; i < grid size; i++)`
`for (j = 2 - i%2; j < grid size; j+=2)`

$$\texttt{u[i, j]=}\ s_5^{-1}(\texttt{f[i, j]} - s_1\texttt{u[i+1, j-1]} + s_2\texttt{u[i+1, j]} + s_3\texttt{u[i+1, j+1]} + s_4\texttt{u[i, j-1]} + s_6\texttt{u[i, j+1]} + s_7\texttt{u[i-1, j-1]} + s_8\texttt{u[i-1, j]} + s_9\texttt{u[i-1, j-1]})$$

update the black points
`for (i = 1; i < grid size; i++)`
`for (j = 1 + i%2; j < grid size; j+=2)`

$$\texttt{u[i, j]=}\ s_5^{-1}(\texttt{f[i, j]} - s_1\texttt{u[i+1, j-1]} + s_2\texttt{u[i+1, j]} + s_3\texttt{u[i+1, j+1]} + s_4\texttt{u[i, j-1]} + s_6\texttt{u[i, j+1]} + s_7\texttt{u[i-1, j-1]} + s_8\texttt{u[i-1, j]} + s_9\texttt{u[i-1, j-1]})$$

The lines labelled `red black(2)` and `red black(4)` were obtained by using a restructured code based on unrolling the the inner `j` loops, see e.g. Fried [5]. This doubles the maximum Mflops rate, but note that it is effective only for grids up to size 256×256 when the problem fits in the cache. Though this code modification does not yet improve the out-of-cache performance, this step is a prerequisite before one goes to the next step to improve the cache behaviour of the algorithm.

4.2. Out-of-Cache Optimisations. To get good out-of-cache performance 2 levels of loop unrolling (instead of 4) turned out to be optimal. Therefore the best in-cache results shown in Figure 2 are not as good as those in Figure 1.

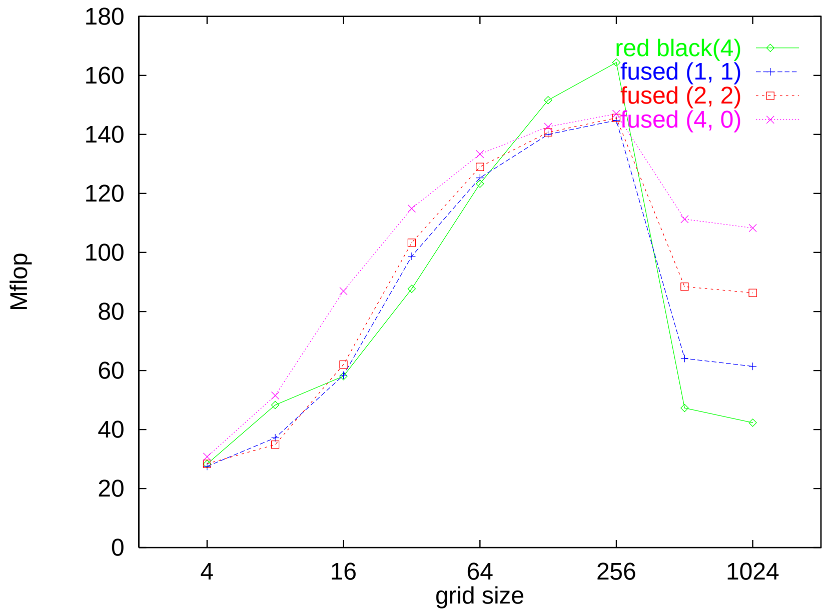

FIGURE 2. Mflops rate for melted versions of red/black Gauss Seidel sweeps.

When implementing the standard red/black Gauss Seidel algorithm the usual practise is to do one complete sweep of the grid updating all of the red nodes and then one complete sweep updating all of the black nodes. The first point to note is that as a red node is updated, the black node directly underneath it may be updated. Consequently, we work in pairs of rows; once all of the red nodes in one row have been updated, all of the black nodes in the previous row are updated. So instead of doing a red sweep and then a black sweep we just do one sweep of the grid updating the red and black nodes as we move through. The line labelled `fused(1,1)` in Figure 2 shows how this can improve the Mflops rate.

In the next step we melt the sweeps together. For example, instead of doing two sweeps, and updating the nodes once during each sweep, we do one sweep and update the nodes twice. Of course this must be done respecting the data dependencies. For a red/black smoother we find that generally a whole stack of k nodes $(i, j) \ldots (i-k, j)$ can be updated within a single sweep. Such a sweep operating on k nodes in a single pass can then replace k successive sweeps which would update evey second node just once. This is illustrated in Figure 3. Coding these algorithms in standard languages is technically rather complicated, especially since the boundaries need a separate treatment.

The line labelled `fused(2,2)` in Figure 2 shows the result of two calls to the melted algorithm with 2 melted sweeps. This corresponds to how the smother could be used in a a V-cycle with two sweeps of pre- and two sweeps of post-smoothing. The line labelled `fused(4,0)` in Figure 2 shows the performance when melt 4 sweeps are melted together. This could be used in a V-cycle using 4 sweeps of pre-smoothing and no post-smoothing. Though this need the same amount of work as the runs for

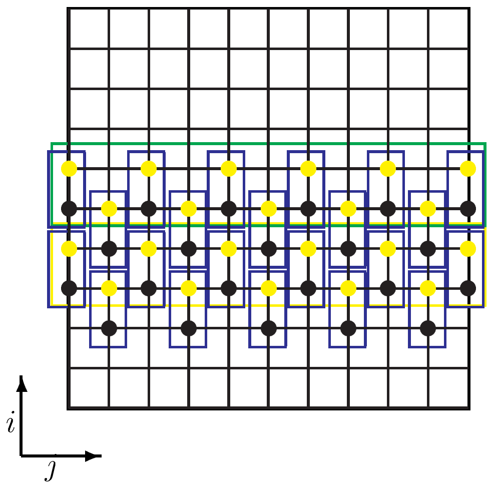

FIGURE 3. The red and black nodes can be updated in pairs.

`fused(2,2)`, it clearly shows better out-of cache performance, when considering the performance for large grids with more than 500×500 nodes. Thus the loop-fusing is clearly effective in improving the data-locality and therefore the potential machine performance.

5. V-cycle and FMV algorithm

Smoothing by a red-black Gauss-Seidel smoother as a core component of the multigrid algorithm has already been discussed in Section 4. For the V-cycle, we have to implement additionally the calculation of the residual, restriction of the residual, interpolation, and correction. All these modules are first optimised for in-cache performance using techniques like loop-unrolling, similar as it has been done for the smoother. Out of-cache performance, however, can only be improved by melting these operations together as far as possible. Again this is similar to the melting of sweeps as has been done for the application of the smoother, however, it is technically more complicated since different operations must be handled.

The multigrid modules can be grouped into pre coarse grid operations (smoother, residual calculation and restriction) and post coarse grid operations (interpolation and smoother). To improve the out-of-cache performance we melted the pre coarse grid operations together and the post coarse grid operations together. Performance results for variations of this technique are shown in Figure 4. Here plots labelled `melt(`μ_1, μ_2`)` represent the number of pre and post smoothers. The graph labelled *optimised RB* shows for comparison the best result obtained without using a melting technique.

For the FMV-scheme we need in addition to all previous components the interpolation operator $\widetilde{I}_{k-1}^{k}$. From the classical multigrid theory [2] it is known that if $\widetilde{I}_{k-1}^{k}$ is of sufficiently high order, a fixed number of V-cycles is sufficient to compute a solution to the level of truncation errors. In our case, to maintain the $O(h^4)$ accuracy offered by the compact 9-point stencil, it is necessary to use a polynomial interpolation of

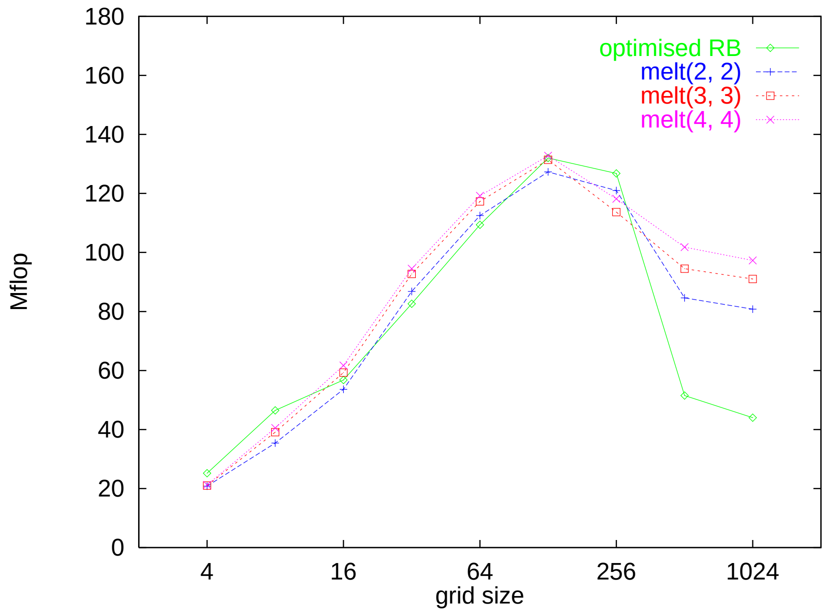

FIGURE 4. Mflops rate for different implementations of the V-cycle.

degree 4. Again this operation could be melted with the other pre-coarse grid operations on the new finest grid. However, our experiments indicate that this results in only a minor further performance improvement.

6. Numerical Results

The 5-point stencil is often preferred over the 9-point discretisation because of its simplicity and smaller operation count. In our context, the compact 9-point stencil offers a higher Mflops rate and a better accuracy for smooth functions. To demonstrate this we solve Poisson's equation on the unit square with boundary conditions and right hand side such that $\sin(\sinh(3x)) * \sin(\sinh(3y))$ is the exact solution. Figure 5 displays the accuracy relative to the CPU time. The comparison of the standard 5-point discretisation and the 9-point discretisation clearly demonstrates that the higher order method is more efficient in this example. Additionally, most of the higher operation count of the 9-point stencil is compensated by the better Mflop rate. However, we use V-cycles (instead of the FMV algorithm) so that the better approximation properties of the discretisation can only be exploited when the number of cycles is increased appropriately, as already done in the experiment shown.

7. Conclusions

This article demonstrates techniques, how the performance of iterative solvers can be increased by improving the cache-awareness of standard algorithms. Though the techniques were designed for structured grids and multilevel algorithms only, we believe that they have wider applicability. The core modules are similar in many iterative solvers. Even for unstructured grids, similar techniques could be employed.

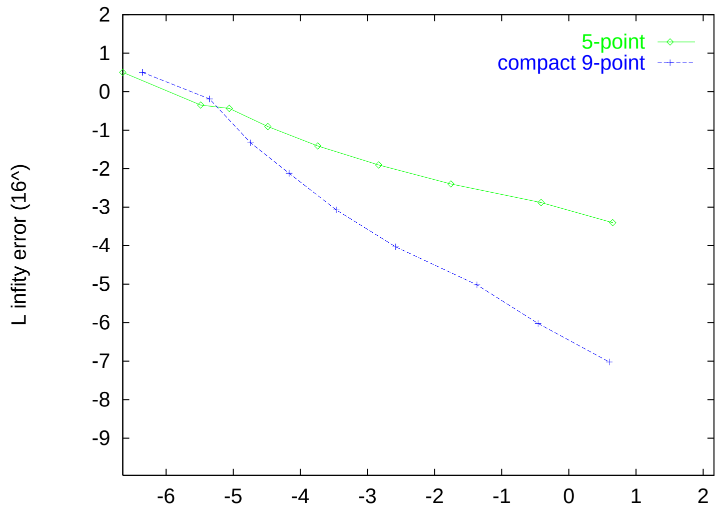

FIGURE 5. Error v's time for test problem using the V-cycle

Our present experience, however, indicates that the more complex data structures and the indirect addressing necessary for implementing unstructured meshes creates a serious performance bottleneck, putting unstructured applications at a serious disadvantage. Our current research is therefore directed at developing a compromise based on patch-adaptive refinement that can on the one side accommodate the need for adaptivity, but which on the other side still permits efficient processing.

Acknowledgement. This research is supported by the Deutsche Forschungsgemeinschaft, project Ru 422/7-1.

References

[1] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Math Comput.*, 31(138):333–390, April 1977.

[2] A. Brandt. *Multigrid techniques: 1984 guide with applications to fluid dynamics.* GMD-Studien, Nr. 85. Gesellschaft für Mathematik und Datenverarbeitung MBH, Bonn, May 1984.

[3] D. C. Burger, J. R. Goodman, and A. Kägi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical Report TR-95-1261, University of Wisconsin, Dept. of Computer Science, Madison, 1995.

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

[5] S. Fried. A tutorial on NDP Alpha compiler driver and optimizations http://www.microway.com/optimizations.html, December 1996.

[6] W. Hackbusch. *Multigrid methods and applications.* Springer Verlag, Berlin, 1985.

[7] S. F. McCormick. *Multigrid Methods.* SIAM Frontiers In Applied Mathematics, 1987.

[8] D. Patterson. *Microprocessors in 2020.* Scientific American, September 1995.