FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

A Guide to Designing Cache Aware Multigrid Algorithms

C.C. Douglas and U. Rüde and J. Hu and M.L. Bittencourt

Lehrstuhlbericht 07-3

A GUIDE TO DESIGNING CACHE AWARE MULTIGRID ALGORITHMS

Craig C. Douglas[1]
University of Kentucky
Departments of Mathematics and Mechanical Engineering
Center for Computational Sciences
715 Patterson Office Tower
Lexington, KY 40506-0027, USA

Ulrich Rüde[2]
Institut für Mathematik
Universität Augsburg
D-86135 Augsburg, Germany

Jonathan Hu and Marco Bittencourt
University of Kentucky
Department of Mathematics
715 Patterson Office Tower
Lexington, KY 40506-0027, USA

SUMMARY

Multigrid methods use a combination of numerical procedures. Typical components include one or more iterative procedures (e.g., Gauß-Seidel relaxation), residual computation, projection, and interpolation. Standard implementations combine the components using a structured programming methodology where each component is programmed separately and called by a multilevel procedure one at a time.

Computers today have a multilevel memory hierarchy that include a memory subsystem known as a cache. Parts of main memory are duplicated inside the cache. Re-using the data in cache can speed up a program by a factor of $6 - 10$ on many computers.

In this paper we define a collection of cache aware, tiled algorithms that make up standard multigrid methods. We show how to implement cache aware algorithms in a manner that is straightforward, easy, and portable. While following the definitions in this paper will not result in getting every last floating point operation per second from any computer, the performance is much better than using standard implementations.

The algorithms we propose appear to use completely unstructured, or spaghetti, programming methodology. However, we show that we require a more structured programming methodology than is usually found in well written multigrid codes.

[1]Partially funded by NATO grant CRG 971574 and NSF grants 9707040 and 9721388.
[2]Partially funded by Deutsche Forschungsgemeinschaft, Ru 422/7-1 and NATO grant CRG 971574.

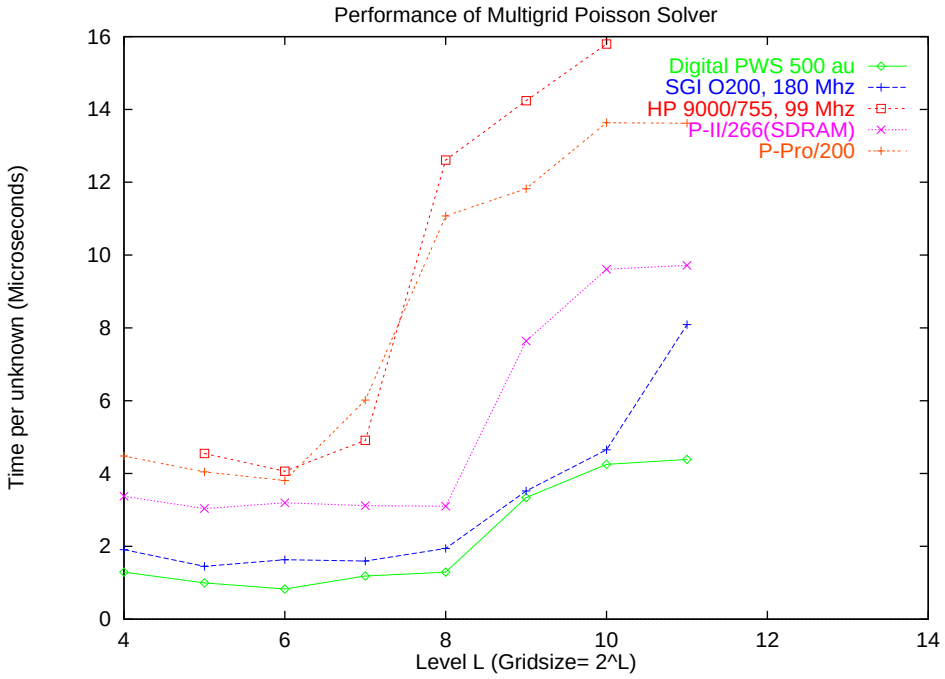

Figure 1: Performance in microseconds per unknown vs. the mesh size

1. INTRODUCTION

Since the early 1980's, computer processors have sped up much faster per year than memory. Multilevel memories, using *memory caches* were developed to compensate for the uneven speed ups in the hardware. Essentially all computers today, from laptops to distributed memory supercomputers use cache memories to keep the processors busy.

By the term cache, we mean a fast memory unit closely coupled to the processor [9]. In the interesting cases, the cache is further divided into a great many *cache lines*, which hold copies of contiguous locations of main memory. The cache lines may hold data from quite separate parts of main memory.

Computer systems commonly have two levels of cache memory. The cost of a second level cache miss is typically more than 100 machine instructions (e.g., on a SGI Indigo2-R4400 with a peak execution rate of 200 MIPS, one main memory access takes 1.17 microseconds). Application performance has become quite sensitive to cache misses as the memory access penalty has increased. This is especially true for applications like multigrid with data sets larger than the cache size that do not exhibit a high degree of memory reference locality.

To demonstrate the performance effect of memory systems, we benchmark a standard multigrid code for solving Poisson's equation on a number of current machine architectures. The example is the first benchmark example suggested in [4]. A multigrid V-cycle with two red-black Gauß-Seidel iterations as the presmoother and one Gauß-Seidel iteration as the postsmoother reaches the benchmark objective of reducing the initial residual by 6 orders of magnitude within 5 cycles. Further, it has a work estimate of less than 250 floating point operations per unknown. The practical performance is measured microseconds per unknown and is presented in Figure 1 for a variety of machines. Instead of requiring a constant time per unknown, there is a clear degradation of performance for larger mesh

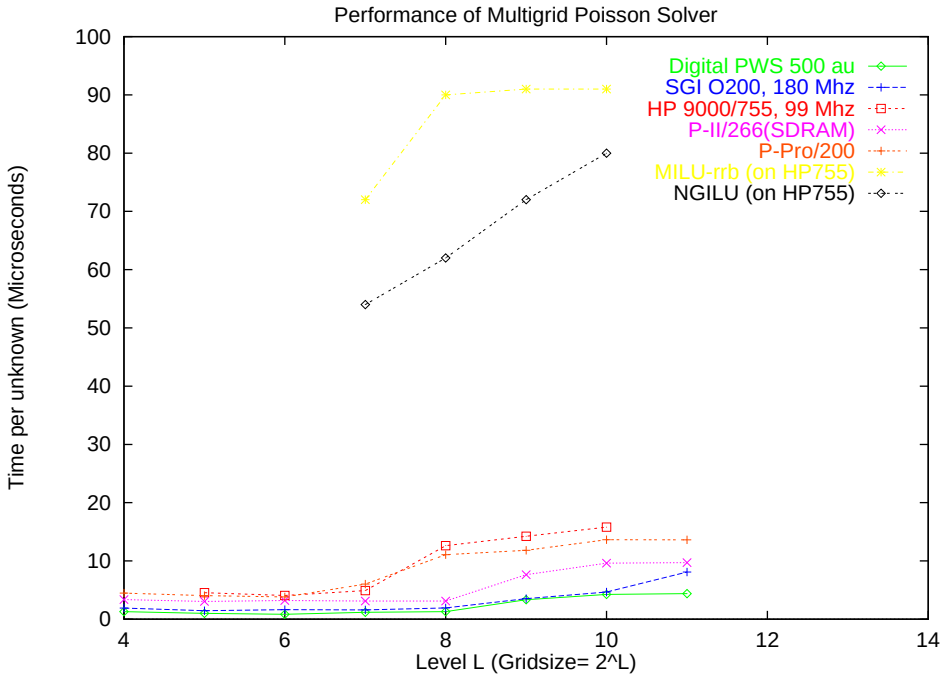

Figure 2: Performance in microseconds per unknown vs. the mesh size

sizes. This, however, is merely a cache effect. The degradation takes place exactly where the data gets so large that it cannot fit into cache any more.

Consider a DEC workstation based on the Alpha 21164 processor operating at 500 Mhz. The best time of less than 1 microecond is obtained for a mesh with 65×65 grid points, where our implementation uses less that 100 KB of memory and completely fits into the on-chip L2 cache. When the size gets larger, the data can still be held in the off-chip level 3 cache up to grid sizes of 256×256, but beyond that the performance degrades to more than 4 microseconds.

The picture is even more disappointing, when considering that the processor is capable of 1 Gflops (i.e.,. 10^9 operations per second), and should thus finish the 250 operations per each unknown in 0.25 microseconds. For large meshes, the Alpha workstation reaches merely 7% of its peak performance. This is not just typical of the Alpha architecture: all other machines show essentially the same picture. The disappointing performance is also not a fault of the multigrid algorithm, as can be seen by a comparison with the original benchmark results in [4]. Figure 2 augments the results from Figure 1 with the best two best performers from [4]. These codes were run on an HP 9000/755. From the description in [4] it is not clear whether the slower performance is due to the preconditioned conjugate gradient algorithms not being competitive with multigrid or because the implementation is slower, or a combination of both. Also note that multigrid should of course be used with nested iteration to solve this problem, reducing the computer time by another factor of 5.

In any case, it is clear that we need to search for methods to speed up the calculations on large grids.

Tiling is the process of decomposing a computation into smaller blocks and doing all of the computing in each block one at a time. Tiling is an attractive method for improving data locality. In some cases, compilers can do this automatically. However, this is rarely the case for realistic scientific codes. In fact, even for simple examples, manual help from

the programmers is, unfortunately, necessary. Furthermore, one may even think to change the algorithm to improve the data locality. In this situation a compiler cannot do the necessary changes automatically.

Language standards interfere with compiler optimizations. Due to the requirements about loop variable values at any given moment in the computation, compilers are not always free to fuse nested loops into a single loop. In part, it is due to coding styles that make very high level code optimization (nearly) impossible [10].

The easiest way to hand tile a multigrid algorithm would be to use a domain decomposition preconditioner with local multigrid solvers to form a block method. We do not do this in this paper since, when done naively, the convergence rate drops to a standard domain decomposition one rather than the very fast rate that the original multigrid method has. Instead, we use sophisticated changes to the basic algorithms used in multigrid to achieve the bitwise exact same answers as a standard multigrid implementation would produce. In this sense we seek for program transformations that may ultimately be done by a compiler.

Among computer scientists today there are two common beliefs about programming for cache effects. The first is that optimizing for one processor system means that the code will work poorly on almost all other systems. Hence, portability is lost. This point of view is only correct when a code is tailored to one processor and its memory subsystem and then tuned as much as is possible.

The second belief is that programming for cache effects is highly intrusive. This view assumes that old style programming techniques are used where code is given and not changeable except by rewriting the code when a new processor comes along. Recent coding techniques, begun in the 1970's, make this belief obsolete.

In this paper, we will attempt to show that by not trying to get every last clock cycle out of a processor and its memory subsystem that we can, in fact, get a portable code that is not intrusive and still gets quite impressive performance on a variety of RISC based microprocessors.

The principal reason that we can do this is that a careful comparison of RISC processors available today show that they have a very large set of common hardware instructions. Memory subsystems, while different from vendor to vendor, share several important features, which we can use for our purposes.

In §2, we discuss how to do cache aware Gauß-Seidel for both the natural and red-black orderings. We compare the standard approach with tiled versions and offer advice on how to construct codes that run faster than would normally be considered. In §3, we consider three components of a multigrid algorithm that are cache aware. In §4, we discuss a nonobvious extension to three dimensions as well as differences between structured and unstructured grid approaches. In §5, we draw some conclusions.

2. RELAXATION METHODOLOGY and NOTATION

In this section we consider modifying the standard Gauß-Seidel implementation to be

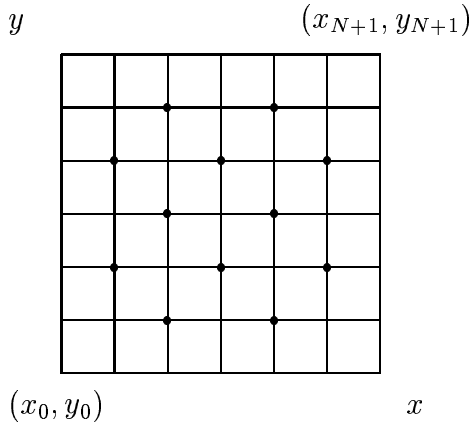

Figure 3: Simple grid with red points marked

cache aware [6]. The techniques used here apply to problems of the form

$$\begin{cases} -\nabla \cdot (a\nabla u) + su &= f \text{ in } \Omega, \\ u &= g_0 \text{ on } \Gamma_0, \\ \dfrac{\partial u}{\partial n} &= g_1 \text{ on } \Gamma_1, \end{cases} \tag{1}$$

where $\Gamma_0 \cup \Gamma_1 = \partial\Omega, \Gamma_0 \cap \Gamma_1 = \emptyset, a(x,y) \geq a_0 > 0$ for some $a_0 \in \mathbb{R}$, and $s(x,y) \geq 0 \; \forall (x,y) \in \Omega$. Unless stated otherwise, we assume that $\Omega \subset \mathbb{R}^2$ is a rectangular domain and that a tensor product or uniform mesh $\mathcal{G}$ covers Ω.

Before transforming the standard Gauß-Seidel algorithms into cache aware versions, let us define two operations for updating the approximate solution on either one or two rows of a grid.

$$Update(\text{ row, color [, direction] })$$

and

$$UpdateRedBlack(\text{ row, [, direction] })$$

We implicitly assume that both *Update* and *UpdateRedBlack* only compute on rows that actually exist. In §3 we argue that the implementation of both of these procedures cannot be subroutines from both an efficiency viewpoint and to avoid unwanted cache side effects.

The operation *Update* does a Gauß-Seidel update in each of the columns of a single row in the grid. The color is one of red, black, or all. The direction is optional and can be natural or reverse. The natural order is assumed unless a direction is given. Hence, symmetric Gauß-Seidel is easily implemented.

The operation *UpdateRedBlack* operates on all of the red points (x_i, y_j) in row j of the grid and on all of the black points $(x_i, y_{j\pm1})$ in the preceding row $j\pm1$ (the updates are paired). The $\pm$ depends on the choice of direction. This fuses the red-black calculation so that we do a red-black ordered Gauß-Seidel with only one sweep across the grid instead of the standard two passes.

The update operations can be further improved upon if we are interested in SOR, SSOR, or ADI relaxation methods. Within the update operations we can further optimize

the process by including Linpack style optimizations like loop unrolling to get $2-7$ updates per iteration through the loop.

Now consider m iterations of Gauß-Seidel, where we mean

Do $i = 1, m$
  1 complete Gauß-Seidel iteration over the entire mesh G.
Enddo

Naturally ordered Gauß-Seidel implementations have certain properties. In the standard implementation, data passes through cache m times. In the tiled implementation, data passes through cache once.

Red-black ordered Gauß-Seidel implementations have certain properties. In the standard implementation, data passes through cache $2m$ times. Further, no compiler on the market automatically tiles even Laplace's equation on a square with a uniform mesh and a five point operator [10]. In the tiled implementation, data passes through cache once.

We now define a tiled version of the naturally ordered Gauß-Seidel. We have to assume that $\ell + m - 1$ rows of a $N \times N$ grid G fit entirely into cache simultaneously and that $m < \ell$.

```
Do it = 0, m − 1
   Do i = 1, ℓ − it
      Update( i, all )
   Enddo
Enddo
Do j = ℓ, N, ℓ
   Do it = 0, m − 1
      j1 = min(j + ℓ, N)
      Do i = j + 1, j1 − it
         Update( i, all )
      Enddo
      Do i = j, j + it − 1, −1
         Update( i, all )
      Enddo
   Enddo
Enddo
```

The first set of loops is for the first ℓ rows only. In each outer iteration a one row buffer is needed in order that the number of updates of each of the neighbors is correct to keep the updates identical to the standard implementation of Gauß-Seidel.

The second set of loops has computation running in two directions. The first inner loop does a set of updates in the natural order for the current block of rows that fit in cache. The second inner loop runs backwards through the rows of part of the previous block of rows that fit in cache. The total number of rows that fit in cache must be large enough so that $\ell + m - 1$ rows fit in cache simultaneously.

The tiled version of the naturally ordered Gauß-Seidel has three very important features. It is easy to implement. It extends trivially to more than two dimensions. Calculating ℓ is simple based on the usable size of cache and N. Typically, the usable part is no more than 50% of the size of the cache. The operating system usually switches tasks often enough so that the rest of cache is not usable by a single task.

We now define a tiled version of the red-black ordered Gauß-Seidel. We have to assume that $\ell+m$ rows of a $N\times N$ grid G fit entirely into cache simultaneously and that $m \leq \ell/2$.

```
Do i = 1, N, ℓ
   j1 = min(j + ℓ, N)
   Do i = j, j1
      Update( i, red )
      Update( i − 1, black )
   Enddo
   Do it = 0, m − 1
      Do i = j − 1, j − m + 2 ∗ it, −2
         If j is odd, then
            UpdateRedBlack( i )
            UpdateRedBlack( i − 1 )
         Else
            UpdateRedBlack( i − 1 )
            UpdateRedBlack( i )
         Endif
      Enddo
   Enddo
Enddo
```

The first set of loops updates the entries in the current block of rows. In each outer iteration a two row buffer that is shaped like a sawtooth blade is needed in order that the number of updates of each of the neighbors is correct to keep the updates identical to the standard implementation of Gauß-Seidel.

The second set of loops has computation running backwards. There are two choices depending on if the row number is even or odd as to where the first red point is in the row closest to the current block. The total number of rows that fit in cache must be large enough so that $\ell + m$ rows fit in cache simultaneously.

This is somewhat harder to implement than the tiled version of the naturally ordered Gauß-Seidel. However, it allows a cheaper version of interpolation (only at the black points) for the multigrid method that we intend to imbed the Gauß-Seidel algorithm into. While it is not obvious, the extension to more than two dimensions can be done in a two dimensional manner very efficiently. For two dimensions performance experiments with these algorithms are reported in [8], [12].

3. COMBINING MULTIGRID COMPONENTS

In this section we define a tiled multilevel algorithm and state conditions for determining how big the blocks are that we use in tiled algorithms. We further state how to implement the algorithms in manner which is both straightforward and easy.

Assume that the finest level and the coarsest level are numbered 1 and k respectively. The V cycle definition relies on three procedures $Correct$, $PreCorrect$, and $PostCorrect$.

The *Correct* procedure is simply one of the tiled Gauß-Seidel routines (with a natural or red-black ordering) doing the equivalent of m iterations.

The *PreCorrect* procedure does the following, all tiled.

- $m-1$ smoothing steps using one of the tiled Gauß-Seidel routines.
- One smoothing step coupled with the residual calculation and projection of the residuals onto the next coarse level's right hand side.

The *PostCorrect* procedure does the following, all tiled.

- $m-1$ smoothing steps using one of the tiled Gauß-Seidel routines.
- One smoothing step coupled with the interpolation and addition of the correction onto the next finer level's approximate solution.

We now now define a tiled version of the V cycle.

Do $level = 1, k-1$
  $PreCorrect(\ level, \cdots\ )$
Enddo
Do $level = k, 2, -1$
  $PostCorrect(\ level, \cdots\ )$
Enddo
$Correct(\ 1, \cdots\ )$

A W or F cycle or a nested iteration multilevel algorithm can be defined just as easily.

Similar algorithms have been described in [6]. Some performance results are contained in [8] and [12].

The implementation of the tiled algorithms in this paper could be quite difficult and problem dependent. However, this does not have to be the case using modern computer science programming techniques. For the simple tensor product grids that we have used so far in the paper, all that is needed are simple *include* statements or a macro preprocessor. The constructs that are needed are to do the smoothing, residual calculation, and projection from a single point. For the interpolation and add step, we need to know how to do the operations from $1-4$ points only.

A highly structured programming style is required in order to make the coding modular. Instead of having a large number of subroutines for major components, we have a number of include files. This is possible only by using the exact same set of indices when referring to any element of any vector. For example, any reference to an element of any level's right hand side $r_j(x_i, y_j)$ would always be programmed as $r(i,j)$ (or whatever convention is required for the programming language in use) no matter what the circumstances. This makes programming a wide variety of problems of the form (1) and different types of domains Ω and grids G (or G_i in the multilevel case).

The *Update* and *UpdateRedBlack* procedures might appear to be separate subroutines. Implementing them in that fashion will completely destroy the potential benefits of using them. They must be either included into the Gauß-Seidel procedures or macro definitions that expand code in line. As separate subroutines they would take more computer time and would cause significant overhead as program stack frames are initialized on entry and deleted on exit, causing unwanted cache side effects.

In order to efficiently do loop unrolling and/or tiling, we need a small amount of information about the computer that we are going to use. First, we need to know how big the usable part of the cache actually is. Knowing the size of usable cache and N we calculate

- ℓ_1 rows fit into cache for $Correct$.
- ℓ_2 rows fit into cache for $PreCorrect$.
- ℓ_3 rows fit into cache for $PostCorrect$.

The second item we need to know is how many points to unroll in the loops in the $Update$ and $UpdateRedBlack$ code.

In an ideal environment, we need an automatic tool to produce all of the stencil updates based on just one stencil centered at some point. Since not all programmers use the same computer language, the tool must be flexible enough to produce code in a variety of languages including some or all of C, C++, Java, Fortran, and Pascal. How long the tool takes to produce code in the target language is largely irrelevant as long as the output is both correct and extremely efficient.

4. PROBLEMS IN THREE DIMENSIONS OR ON UNSTRUCTURED GRIDS

Some of the fastest multigrid solvers ever written were produced for a contest to see who would sell NASA a set of parallel computers. The NAS benchmarks, [1]and [2], distinguish themselves because they are given as descriptions on paper of what the final programs are required to do. It is up to the implementor how to implement each code anyway they wish.

One of the NAS benchmarks is a multigrid one. The problem is to solve the Poisson equation on the unit cube $(0,1)^3$ with a periodic boundary condition along two edges of the domain. The finest grid has 256^3 grid points.

When determining how to block a three dimensional grid on a cubic domain for the most efficient re-use of data in the cache, we have two choices. The first is to use small, three dimensional cubes. The second is to extend the two dimensional cache aware algorithms to three dimensions, but optimize only in the planes.

Our experiments so far show that the two dimensional approach works best. As a bonus, coding red-black ordered Gauß-Seidel in this case is both straight forward and easy. Programming red-black ordered Gauß-Seidel on three dimensional subcubes, with a sawtoothed shaped buffer, is almost impossible to do correctly by hand and it is not so obvious that it runs very fast either.

The unstructured grid case in two dimensions provides another very difficult, but interesting case. The number of grid elements that any vertex in the grid is connected to is neither constant nor, in most cases, nearly constant. A matrix derived from the discretization of (1) on an unstructured grid does not have a nice number of nonzeroes per row as we had in the structured grid case of §§2-3. When there is a wide variety in the number of nonzeroes in row of the matrices, loop unrolling is really not applicable.

The unstructured grid case is beyond the scope of this paper. The results will be presented at a later time.

5. CONCLUSIONS

In this paper we have defined a number of tiled, cache aware algorithms that can be used to implement multigrid in cache aware manner. In addition, a programming methodology was proposed that is both straight forward and easy to implement, but is also portable as long as the usable cache size and the size of the grids are known.

The convergence rate of the multigrid method proposed in this paper is identical to the convergence rate of a standard implementation. Hence, all of theory that can be found in [3], [5], [7], [11], and [13] apply to the algorithms in this paper. This is due to our producing bitwise identical results to the standard implementations.

While the general perceptions are that cache based algorithms are both too intrusive and too machine specific, this is not the case. One reason why these perceptions exist is that high performance computer vendors have produced codes, written in assembly language (or as close to it as they can program), that are exceptionally machine dependent.

There persists the belief that all cache based algorithms will hold the property of not being portable. Most of the high performance microprocessors in use today have very similar, RISC based architectures. Algorithms that work well on one of these processors, but are not completely tuned for a processor, usually work just as well on the rest of the similar processors.

Since memory system designs are not likely to change in the next few years, using cache aware algorithms makes more and more sense as time progresses, not less sense.

REFERENCES

[1] BAILEY, D.H., BARSZCZ, E., BARTON, J.T., BROWNING, D.S., CARTER, R.L., DAGUM, L., FATOOHI, R.A., FINEBERG, S., FREDERICKSON, P.O., LASINSKI, T.A., SCHREIBER, R.S., SIMON, H.D., VENKATAKRISHNAN, V., and WEERATUNGA, S.K.: "The NAS parallel benchmarks (94)", Technical Report RNR–94–007, NASA Ames, Ames, CA, 1994. Available as http://www.nas.nasa.gov/NAS/TechReports/RNRreports/dbailey/RNR-94-007/RNR-94-007.html.

[2] BAILEY, D.H., BARSZCZ, E., BARTON, J.T., BROWNING, D.S., CARTER, R.L., DAGUM, L., FATOOHI, R.A., FINEBERG, S., FREDERICKSON, P.O., LASINSKI, T.A., SCHREIBER, R.S., SIMON, H.D., VENKATAKRISHNAN, V., and WEERATUNGA, S.K.: "The NAS parallel benchmarks", International Journal of Supercomputer Applications, 5, 1991, pp. 63–73.

[3] BANK, R.E. and DOUGLAS, C.C.: "Sharp estimates for multigrid rates of convergence with general smoothing and acceleration", SIAM Journal on Numerical Analysis, 22, 1985, pp. 617–633.

[4] BOTTA, E.F.F., DEKKER, K., NOTAY, Y., VAN DER PLOEG, A., VUIK, C., WUBS, F.W., and DE ZEEUW, P.M.: "How fast the Laplace equation was solved in 1995", Applied Numerical Methods, 24, 1997, pp. 439–455.

[5] DOUGLAS, C.C.: "Multi-grid algorithms with applications to elliptic boundary-value problems", SIAM Journal on Numerical Analysis, 21, 1984, pp. 236–254.

[6] DOUGLAS, C.C.: "Caching in with multigrid algorithms: problems in two dimensions", Parallel Algorithms and Applications, 9, 1996, pp. 195–204.

[7] HACKBUSCH, W.: "Multigrid Methods and Applications", Computational Mathematics 4, Springer-Verlag, Berlin, 1985.

[8] HELLWAGNER, H., WEISS, C., STALS, L., and RÜDE, U.: "Data locality optimizations to improve the efficiency of multigrid methods", this book, 1998.

[9] PATTERSON, D.A. and HENNESSY, J. L.: "Computer Architecture: A Quantative Approach", Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[10] PHILBIN, J., EDLER, J., ANSHUS, O.J., DOUGLAS, C.C., and LI, K.: "Thread scheduling for cache locality", in Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 1996, ACM, pp. 60–73.

[11] RÜDE, U.: "Mathematical and Computational Techniques for Multilevel Adaptive Methods", Frontiers in Applied Mathematics 13, SIAM Books, Philadelphia, 1993.

[12] STALS L., and RÜDE, U.: "Techniques for improving the data locality of iterative methods", Technical Report MRR 038-97, Australian National University, 1997.

[13] WESSELING, P.: "An Introduction to Multigrid Methods", John Wiley & Sons, Chichester, 1992.