Caching in with Multigrid Algorithms: Problems in Two Dimensions

Craig C. Douglas[1]

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY, USA;
Computer Science Department, Yale University, New Haven, CT, USA.

Abstract: Multigrid methods combine a number of standard sparse matrix techniques. Usual implementations separate the individual components (e.g., an iterative methods, residual computation, and interpolation between grids) into nicely structured routines. However, many computers today employ quite sophisticated and potentially large caches whose correct use are instrumental in gaining much of the peak performance of the processors.

We investigate when it makes sense to combine several of the multigrid components into one, using block oriented algorithms. We determine how large (or small) the blocks must be in order for the data in the block to just fit into the processor's primary cache. By re-using the data in cache several times, a potential savings in run time can be predicted. This is analyzed for a set of examples.

Key words: multigrid, cache, sparse matrix, iterative methods, domain decomposition.

1 INTRODUCTION

In this paper, we investigate the effect computer caches (see [13]) can have on tailored multigrid algorithms (see [4]). This topic is timely since larger caches with many cache lines are becoming quite common on computer processors.

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

For example, a cache with $.25 - 4$Mb of cache memory with .25Kb cache lines allows for many interesting possibilities, whereas a processor with 8Kb of cache memory with 2Kb cache lines does not. This is because in the first case many separate lines (or short vectors) can be held in cache simultaneously, whereas in the second case only 4 cache lines are available, which is insufficient for the ideas in this paper.

Multigrid algorithms combine a number of operations in order to work. These include iterative methods (typically relaxation methods), residual computation, projection of residuals onto a coarser grid, and interpolation of corrections onto a finer grid. These are typically programmed as separate routines, which makes the components easy to replace and modify.

However, a number of components re-use data in a manner which is suitable for algorithms which are *cache aware*. We will investigate which of these are really worth combining and which are not.

In §2, a brief description of how a cache operates is presented. In §3, cache aware algorithms are developed and analyzed.

2 A CACHE MODEL

A complete description of all of the types of caches currently in use by the computer industry is beyond the scope of this paper (see [13] for an extensive description). However, a model is developed in this section that will be used to analyze the cache aware algorithms in §3.

Some processors have several levels of caches. In this paper, only the primary cache is studied. This cache is typically accessed at a very high rate, proportional to the cycle time of the processor.

On many high performance processors, a separate cache is maintained for data and processor instructions. The advantage of this is that the size of the data cache is always known instead of having to be estimated dynamically. In either case, the minimum size of the data cache can usually be determined accurately enough for the purposes of this paper.

One of a cache's operations is to determine if an instruction is fetching data from a part of main memory that is not duplicated in one of its cache lines (this is referred to as a *cache miss*). In this case, a small portion of the main memory is copied into a cache line. Some caches are designed such that the critical datum is moved first to the cache. This means that the instruction that caused the cache miss can restart sooner than when the whole cache line has to be moved before a restart.

When a cache miss is serviced, the data that is already in the cache line that will be used must be moved back to the main memory. Many caches are designed with a set of temporary cache lines (referred to as *write-back cache lines*) whose only function is to store an existing cache line before the write to main memory is completed. There are currently two common methods for choosing which cache line is replaced: least recently referenced and random. Clearly, the former is preferable to the latter, but is more expensive to implement in hardware.

A typical model for how long a cache miss takes to completely service is given by

$$T_{miss} \leq T_0 + T_1(n) + T_2(n).$$

T_0 is effectively the time to determine which cache line to use and possibly move the data already in that line to a write-back cache line. $T_1(n)$ is the time to move n words of memory from main memory to the cache line. $T_2(n)$ is the amount of time to actually move the data back to main memory. T_2 is usually done by stealing cycles from the memory subsystem. On processors with write-back cache lines, $T_2(n)$ will not necessarily always be visible to the user depending on what is done after the cache miss.

This model is very similar to message passing models for parallel computers. Note that T_0 corresponds to the latency time of initiating a message and that T_1 and T_2 correspond to the data transfer time for long messages.

Example 1: Suppose a machine has a 4 word wide memory bus, cache lines of 32 words, and an adequate number of write-back cache lines. Suppose T_0 is in the range 8–12 cycles (in the ensuing examples, $T_0 = 8$). $T_2(32)$ is 8 or more cycles, depending on the cycle stealing properties of the processor. However, $T_1(32)$ is at most 8 cycles, depending on how the critical word is processed. At a minimum, 16 cycles are required in order to vacate and fill a cache line. On many high performance processors, a total of C_{int} integer and C_{fp} floating point operations can be completed in each cycle, where

$$1 \leq C_{int},\ C_{fp} \leq 4$$

typically. Hence, the first 16 cycles used to service a cache miss could have been spent doing up to $16-64$ floating point and $16-64$ integer operations. While some of these cycles will presumably be spent computing after the critical word is transferred (assuming that the processor has this ability), a number of cycles will not be used for computing. Further, accesses to main memory is temporarily limited by the 8 cycles of writing to main memory implied by $T_2(32)$. Thus, a quite substantial penalty is paid for a cache miss. □

Algorithm Vcycle(k, $\{A_i, u_i, f_i, r_i\}_{i=1}^{k}$)
1. Do $i = 1, 2, \cdots, k-1$
   - 1a. Approximately solve $A_i u_i = f_i$.
   - 1b. Compute a residual $r_i \leftarrow f_i - A_i u_i$.
   - 1c. Set $f_{i+1} \leftarrow R_i r_i$ and $u_{i+1} \leftarrow 0$.
   - 1d. End Do
2. Do $i = k, k-1, \cdots, 1$
   - 2a. Approximately solve $A_i u_i = f_i$.
   - 2b. If $i > 1$, then set $u_{i-1} \leftarrow u_{i-1} + P_i u_i$.
   - 2c. End Do

Figure 1: V cycle definition

No matter how large a cache is, the number of words in each cache line must be proportional to the width of the memory bus. The cache is well balanced if a cache line can be filled from or stored back to memory in a small number of processor cycles.

3 CACHE AWARE ALGORITHMS

In this section, a multigrid algorithm (see [4], [5], [9], and [10]) is transformed into a set of cache aware algorithms. An analysis is provided which shows how useful this is for reducing computing time.

Consider solving the following set of problems:

$$A_i u_i = f_i, \qquad 1 \leq i \leq k,$$

where $u_i \in \mathbb{R}^{n_i}$. In particular, $n_i > n_{i+1}$. Hence, level 1 is the real problem; all other levels are approximations to this. The linear systems result from discretizing a partial differential equation over a given grid Ω_i. The discretization can be any standard finite element, difference, volume, or wavelet approach (see [3] and [8] for examples of this approach). In this paper, A_i is assumed to be a sparse matrix.

A typical multigrid method is based on a V cycle multigrid method is given in Figure 1. Implementing a W Cycle or some hybrid is a straight forward extension to this definition.

There are 3 major operations in this algorithm:

1. Approximate solves: steps 1a and 2a. This is typically a relaxation method for simple problems, but can be any iterative method.

2. Restriction of residuals: steps 1b and 1c. This is typically a weighted average of nearby residuals. It is used to compute a residual correction problem's right hand side (represented by the operator R_i) on the next coarser grid.

3. Prolongation of corrections: step 2b. This is typically a second or fourth order interpolation method (represented by the operator P_i).

In a typical multigrid code, a V cycle is implemented very similarly to the description given here. By using a structured language methodology, different algorithms can be substituted for the default ones trivially. The point of this section is to show that performance can be improved by using a *spaghetti programming* technique.

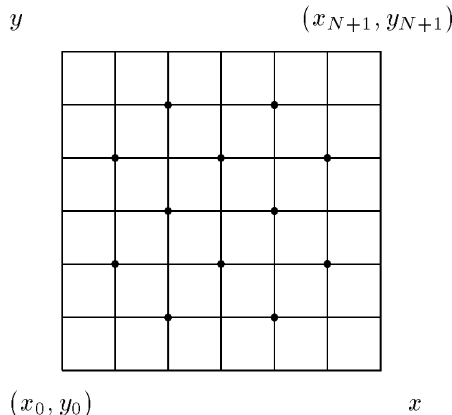

Figure 2: Simple grid with red points marked

3.1 APPROXIMATE SOLVER

Consider just one aspect of the V cycle, namely the approximate solve in steps 1a and 2a. A popular approximate solver is Gauss-Seidel with either the natural or red-black ordering. The red-black ordering has the advantage that it parallelizes in a nice way (communication reduces to sending half of the boundary information at a time which allows for overlapping communication and computing). On both serial and parallel computers it also reduces the cost of the prolongation procedure since only the black points need to be corrected. See [6], [7], [11], [12] for more details.

Gauss-Seidel with either of these orderings can be made cache aware, but will no longer necessarily have the exact same ordering. Consider the grid in Figure 2, where the boundary points are included in the grid.

The usual red-black ordered Gauss-Seidel iteration performs Gauss-Seidel on all of the red points and then all of the black points. The algorithm can be translated into the following:

1. Update all of the red points in row 1.
2. Do $j = 2, N$
   - 2a. Update all of the red points in row j.
   - 2b. Update all of the black points in row $j-1$.
   - 2c. End Do
3. Update all of the black points in row N.

When 4 grid rows of data (u_i and f_i) along with the information from the corresponding rows of the matrix A_i can be stored in cache simultaneously, this is a cache based algorithm. The advantage is that all of the data and the matrix pass through cache only once instead of the usual twice. As shown in Example 1, this may be a considerable savings in time. In fact, the savings percentage is obviously

$$\left(1 - \frac{T_{comp} + T_{cm}}{T_{comp} + 2T_{cm}}\right) \times 100\%,$$

where T_{comp} is the total time spent on computing and T_{cm} is the total time spent on cache misses.

When the grid gets too large, a change in the algorithm is required which changes the global ordering. In essence, we use a domain decomposition approach to find disjoint two dimensional

subdomains Ω_{ij} whose union is Ω_i. Further, the data associated with the Gauss-Seidel operation on each subdomain must fit entirely in cache. The size of the subdomains depends heavily on the number of nonzeroes per row in the matrix A_i and the sparse matrix storage method.

Consider two examples. Both assume that there are 1024 cache lines capable of holding 32 data items each. Hence, 32 entries of each of u_i and f_i fit in a cache line, assuming that they are stored as separate vectors. While a red-black ordering is used inside each subdomain, this is not the global red-black ordering. Both examples assume processor and cache speeds similar to those in Example 1. Hence, if all 1024 cache lines are used while computing on a subdomain Ω_{ij}, then the approximate number of cycles spent on cache misses ranges from 16384 (16 cycles per miss) to 24576 (24 cycles per miss).

Example 2: In this example, assume that A_i is a five point operator with the nonzeroes stored next to each other in an orderly fashion. Hence, only 6 rows of A_i fit in a cache line, which becomes the limiting factor in determining how big Ω_{ij} can be. Then each Ω_{ij} is limited to $(1024/8)\times 32 = 4096$ points, or a 64×64 subgrid.

The update to each u_i involves only 5 multiplies and 4 adds. Hence, this can be done in about $5 - 9$ cycles. Approximately $20480 - 36864$ cycles are spent computing the updated u_i on a 4096 point Ω_{ij}. Hence the savings of the cache based algorithm over the usual one is given by

Cycles per miss	Cycles computing per point		
	1	5	9
16	44%	24%	36%
24	46%	29%	35%

□

Example 3: In this example, assume that the entries of A_i are constants which do not vary over the rows of A_i corresponding to the unknowns (e.g., a Poisson or constant coefficient Helmholtz equation). Then each Ω_{ij} is limited to $(1024/2)\times 32 = 16384$ points, or a 128×128 subgrid.

The update to each u_i involves only 1 multiply and 4 adds. Hence, this can be done in about $3 - 5$ cycles. Approximately $12288 - 20480$ cycles are spent computing the updated u_i on a 16384 point Ω_{ij}. Hence the savings of the cache based algorithm over the usual one is given by

Cycles per miss	Cycles computing per point		
	1	3	5
16	33%	36%	36%
24	38%	30%	35%

□

For both of these examples, the savings percentage is maximized for the unlikely choice of 1 cycle computing per point. The more realistic choice of $1 - 2$ floating point operations per cycle gives a better idea of the savings (this corresponds to the last two columns in the tables).

3.2 V CYCLE

Let us reconsider the V cycle algorithm found in Figure 1. Steps 1a and 2a are approximate solution procedures. Usually, this is just a small (≤ 4) number of iterations of a relaxation method. The preferred one for simple elliptic partial differential equations is Gauss-Seidel.

Consider steps 1a and 2a of the V cycle first. Assume that m_i iterations of the locally red-black Gauss-Seidel are done in each of these steps.

We begin with all of step 2. By the usual approach, A_i, u_i, and f_i would each pass through the cache $2m_i$ times by the end of the approximate solve step 2a. By using a cache aware method,

Algorithm Cache-Vcycle(k, $\{A_i, u_i, f_i, r_i\}_{i=1}^{k}$)

1. Do $i = 1, 2, \cdots, k-1$
   - 1a. Do $\ell = 1, 2, \cdots, m_i$
     - 1a1. Determine $\Omega_{ij}^{(\ell)}$.
     - 1a2. For each $\Omega_{ij}^{(\ell)}$,
       - 1a2a. If $\ell = 1$, then $u_i \mid_{\Omega_{ij}^{(\ell)}} \leftarrow 0$.
       - 1a2b. Do 1 iteration of approximate solve of $A_i u_i = f_i$.
       - 1a2c. If $\ell = m_i$, then compute as much of r_i as is possible.
     - 1a3. If $\ell = m_i$, then complete the calculation of r_i and calculate $f_{i+1} = R_i r_i$.
     - 1a4. End Do
   - 1b. End Do
2. Do $i = k, k-1, \cdots, 1$
   - 2a. Do $\ell = 1, 2, \cdots, m_i$
     - 2a1. Determine $\Omega_{ij}^{(\ell)}$.
     - 2a2. For each $\Omega_{ij}^{(\ell)}$,
       - 2a2a. If $\ell = 1$ and $k = 1$, then $u_i \mid_{\Omega_{ij}^{(\ell)}} \leftarrow 0$.
       - 2a2a. If $\ell = 1$ and $k > 1$, then $u_i \mid_{\Omega_{ij}^{(\ell)}} \leftarrow u_i \mid_{\Omega_{ij}^{(\ell)}} + P_{i+1} u_{i+1} \mid_{\Omega_{ij}^{(\ell)}}$.
       - 2a2c. Do 1 iteration of approximate solve of $A_i u_i = f_i$.
     - 2a3. End Do
   - 2b. End Do

Figure 3: Cache aware V cycle definition

outlined earlier for the Gauss-Seidel iteration, A_i, u_i, and f_i pass through cache only m_i times each. During step 2b, u_i and u_{i-1} would pass through cache once each. Using the cache aware approach, only u_{i-1} would pass through cache as an extra data movement since u_i would already be there from step 2a.

Combining steps 2a and 2b, while possible, is not always reasonable. When the elements of A_i can be guaranteed to be the parts of cache replaced by u_{i-1}, then this is a useful technique. However, this is highly unlikely to be the case in general. In the case of Example 3, it is impossible since there is no A_i in cache.

Assume that $n_{i-1}/n_i \approx 4$, as is usual with two dimensional multigrid examples. Since room in cache must be saved for u_{i-1}, the number of grid points that are in the Ω_{ij}'s is reduced considerably. In Example 2, the reduction is by 4/11 and in Example 3, it is by 2/3. A more general approach is to include step 2b as part of the initialization part of the approximate solver in step 2a.

Now consider all of step 1. Using the cache aware approach, step 1a can be modified to include the initialization of u_i that would normally be done in steps 1c or 2b. At the end of step 1a we can calculate the residual in most of each Ω_{ij}, but not all. Hence, for almost all of the domain Ω_i, the matrix A_i can be re-used in cache one more time than might be expected. Once the residuals are

Algorithm DDM-Cache-Vcycle(k, $\{A_i, u_i, f_i, r_i\}_{i=1}^{k}$)
1. Do $i = 1, 2, \cdots, k-1$
   1a. Within each Ω_{ij},
      1a1. Approximately solve $A_i u_i = f_i$ in Ω_{ij}, including setting $u_i \leftarrow 0$ if $i < k$.
      1a2. Compute a residual $r_i \leftarrow f_i - A_i u_i$.
      1a3. Set $f_{i+1} \leftarrow R_i r_i$.
   1b. End Do
2. Do $i = k, k-1, \cdots, 1$
   2a. Within each Ω_{ij},
      2a1. Approximately solve $A_i u_i = f_i$ in Ω_{ij}, including setting $u_i \leftarrow u_i + P_{i+1} u_{i+1}$ if $i > 1$.
   2b. End Do

Figure 4: Domain decomposition, cache aware V cycle definition

calculated, the projection step 1c can be done. Hence, most of the residuals need only be in cache once, not the expected twice.

Assume that m_i steps of the locally red-black ordered Gauss-Seidel are done. By the usual approach, A_i, u_i, and f_i would each pass through the cache $2m_i + 1$ times by the end of the residual calculation (steps 1a and 1b). The residual would pass through cache twice by the end of step 1c. By using a cache aware method, outlined earlier for the Gauss-Seidel iteration, A_i, u_i, and f_i pass through cache only m_i times and the residual r_i is in cache once.

The cache aware version of Algorithm Vcycle from Figure 1 is given in Figure 3. This at first appears much more complicated than the original, noncache aware algorithm. Surprisingly, it is not much more complicated to program, however.

The subdomain concept is carried further in Figure 3 than has been used so far in this paper. The number of vectors in cache changes as a function of the iteration of the approximate solver. It is quite easy to compute the sizes of the $\Omega_{ij}^{(\ell)}$ before the computation begins, so that steps 1a1 and 2a1 in Figure 3 can be implemented simply as looking up loop bounds.

Consider step 2. In the first iteration, step 2a2a requires a small part of u_{i-1} to be in cache. This is true only when $k > 1$ and $l = 1$. Otherwise, only A_i, u_i, and f_i are required to be in cache.

Now, what is the relative cost of each component of the V cycle? For Examples 2 and 3, this can be determined fairly easily assuming that R_i is a standard nine point weighting and that P_{i+1} is bilinear interpolation onto the black points.

Operation	Passes through cache	(multiplies, adds) per subdomain point	
		Example 2	Example 3
Approximate Solve	A_i, u_i, f_i	$(4, 5)$	$(1, 5)$
Residual	A_i, u_i, f_i, r_i	$(5, 5)$	$(1, 5)$
Restriction	r_i, f_{i+1}	$(.5, 2)$	$(.5, 2)$
Prolongation	u_i, u_{i-1}	$(.25, 1.5)$	$(.25, 1.5)$

Hence, the major cost will be in the approximate solve and residual computations, not the grid transfers. However, reducing cache misses is still a key point.

3.3 DOMAIN DECOMPOSITION

Suppose a true domain decomposition approach is taken during the approximate solve steps 1a and 2a. By this, all m_i iterations of the approximate solver are taken on one subdomain after another in some order. This is just a multiplicative domain decomposition algorithm. Since the data all fits in cache, then A_i, u_i, f_i, and r_i are each in cache only once. The domain decomposition, cache aware version of Algorithm Vcycle from Figure 1 is given in Figure 4.

The effect on the number of cache misses is quite dramatic. The reduction in cache hits per type of data over the usual implementations is given by the following:

Type of data	Cache – VCycle		DDM – Cache – VCycle	
	Step 1	Step 2	Step 1	Step 2
A_i	$\frac{m_i+1}{2m_i+1}$	.5	$\frac{2m_i}{2m_i+1}$	$\frac{2m_i-1}{2m_i}$
f_i	$\frac{m_i+1}{2m_i+1}$	.5	$\frac{2m_i}{2m_i+1}$	$\frac{2m_i-1}{2m_i}$
u_i	$\frac{m_i+1}{2m_i+1}$	$\frac{m_i+1}{2m_i+1}$	$\frac{2m_i}{2m_i+1}$	$\frac{2m_i}{2m_i+1}$
r_i	.5	–	.5	–

Recalling that m_i is typically 2 or 4, these reductions are quite significant.

4 CONCLUSIONS

It is surprising to see how large a subdomain can fit into a relatively small, well designed cache (e.g., 256Kb). As caches continue to increase in size, the ideas of this paper will extend quite nicely to three dimensional problems. While most of the savings in time are with respect to the approximate solver, using a multiplicative domain decomposition method saves even more time and allows us to use theoretical convergence rates from that field for our problems.

ACKNOWLEDGMENTS

The author would like to thank Jim Shearer and Uli Rüde for helpful discussions. The author became interested in this topic while working on IBM's multigrid entry for the NAS benchmarks (see [1] and [2]) in March, 1994.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, S. Fineberg, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks (94). Technical Report RNR–94–007, NASA Ames, Ames, CA, 1994. To obtain the latest version of this document, send e-mail to npb@nas.nasa.gov.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *Inter. J. Supercomputer Appl.*, 5:63–73, 1991.

[3] R. E. Bank and C. C. Douglas. Sharp estimates for multigrid rates of convergence with general smoothing and acceleration. *SIAM J. Numer. Anal.*, 22:617–633, 1985.

[4] A. Brandt. Multi–level adaptive solutions to boundary–value problems. *Math. Comp.*, 31:333–390, 1977.

[5] W. L. Briggs. *A Multigrid Tutorial.* SIAM Books, Philadelphia, 1987.

[6] W. L. Briggs, L. Hart, S. F. McCormick, and D. Quinlan. Multigrid methods on a hypercube. In S. F. McCormick, editor, *Multigrid Methods: Theory, Applications, and Supercomputing*, volume 110 of *Lecture Notes in Pure and Applied Mathematics*, pages 63–83. Marcel Dekker, New York, 1988.

[7] T. F. Chan and R. S. Tuminaro. Design and implementation of parallel multigrid algorithms. In S. F. McCormick, editor, *Proceedings of the Third Copper Mountain Conference on Multigrid Methods*, pages 101–115, New York, 1987. Marcel Dekker.

[8] C. C. Douglas. Multi–grid algorithms with applications to elliptic boundary–value problems. *SIAM J. Numer. Anal.*, 21:236–254, 1984.

[9] C. C. Douglas and J. Douglas. A unified convergence theory for abstract multigrid or multilevel algorithms, serial and parallel. *SIAM J. Numer. Anal.*, 30:136–158, 1993.

[10] C. C. Douglas, J. Douglas, and D. E. Fyfe. A multigrid unified theory for non-nested grids and/or quadrature. *E. W. J. Numer. Math.*, 2:285–294, 1994.

[11] S. F. McCormick. *Multigrid Methods*, volume 3 of *Frontiers in Applied Mathematics.* SIAM Books, Philadelphia, 1987.

[12] S. V. Parter. Estimates for multigrid methods based on red–black Gauss–Seidel smoothings. *Numer. Math.*, 52:701–723, 1988.

[13] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.