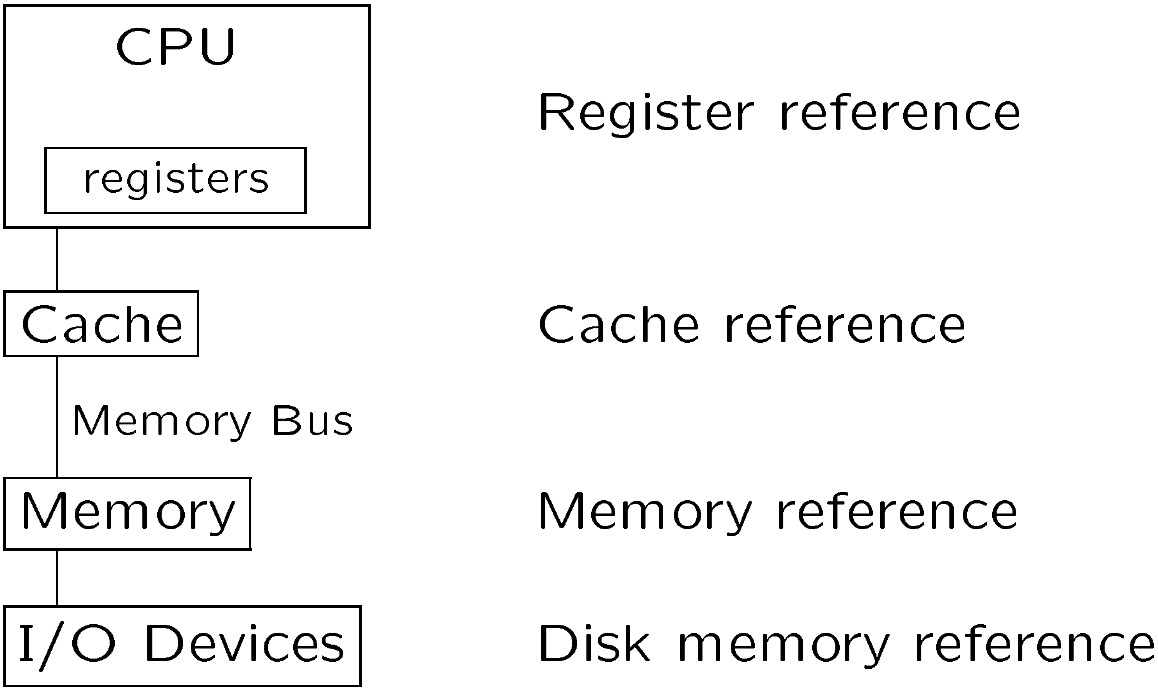
Caching in with Multigrid Algorithms: Problems in Two Dimensions

Dr. Craig C. Douglas

IBM T. J. Watson Research Center (Math)
Yale University (Computer Science)



These are the levels in a typical memory hierarchy. As we move further away from the CPU, the memory in the levels becomes larger and slower.

1. Introduction

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 CPU contains at least one *cache level*. This may be integrated into the CPU chip as SRAM or as a separate set of chips in a CPU chip set.

CPU's with up to 3 levels of caches have been built so far. Multiple caches can slow processing under a number of circumstances.

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.

Multigrid algorithms combine a number of operations in order to work.

1. Numeric solver(s)

2. Interpolation

3. Residual projection

These are typically programmed as separate routines. A number of components re-use data in a manner which is suitable for algorithms which are *cache aware*.

How many of these components should be combined is a function of how good or bad the cache design on a machine is. The worse the design, the more components that can be profitably combined.

2. A Cache Model

Some processors have several levels of caches. Here, only the primary cache is studied.

On many processors, a separate cache is maintained for data and processor instructions.

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 (a *cache miss*). Some caches are designed such that the critical datum is moved first to the cache.

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 (*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*.

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, $T_0 = 8$). $T_2(32) \geq 8$ and $T_1(32) = 8$. Minimumally, 16 cycles are required in order to vacate and fill a cache line. On many processors, a total of C_{int} integer and C_{fp} floating point operations can be completed in each cycle, where

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

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.

3. Cache Aware Algorithms

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, $n_i > n_{i+1}$. Level 1 is the real problem. The linear systems result from discretizing a PDE over given grids Ω_i. The discretization can be any standard finite element, difference, volume, or wavelet approach. Here, A_i is assumed to be a sparse matrix.

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

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

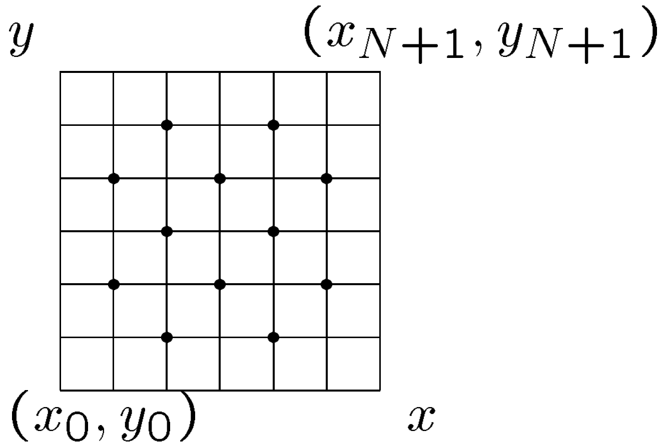
1. Do $i = 1, 2, \cdots, k-1$
   - 1a. Approximately solve $A_i u_i = f_i$.
   - 1b. Compute a residual $r_i \leftarrow f_i - A_i u_i$.
   - 1c. Set $f_{i+1} \leftarrow R_i r_i$ and $u_{i+1} \leftarrow 0$.
   - 1d. End Do
2. Do $i = k, k-1, \cdots, 1$
   - 2a. Approximately solve $A_i u_i = f_i$.
  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

There are 3 major operations in this algorithm: approximate solves (steps 1a and 2a), and restriction of residuals (steps 1b and 1c), prolongation of corrections (step 2b).

Typical multigrid codes are implemented very similarly to the description given here. Performance can be improved by using a *spaghetti programming* technique.

3.1 Approximate Solver

A popular approximate solver is Gauss-Seidel with either the natural or red-black ordering (red points highlighted).



The red-black ordering has the advantage that it parallelizes in a nice way allowing overlapped 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.

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. All of the data and the matrix pass through cache only once instead of the usual twice.

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, we use a domain decomposition approach to find disjoint two dimensional subdomains Ω_{ij} whose union is Ω_i. The data on each subdomain must fit entirely in cache.

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%

3.2 V Cycle

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.

All of step 2

Normally, 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 instead of m_i times as in the cache aware approach.

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. In Example 3, it is impossible since there is no A_i in cache. A more general approach is to include step 2b as part of the initialization part of the approximate solver in step 2a.

All of step 1

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. Once the residuals are calculated, the projection step 1c can be done. Hence, most of the residuals need only be in cache once.

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.

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 $\ell = 1$ & $i > 1$, then $u_i \left|_{\Omega_{ij}^{(\ell)}} \leftarrow 0\right.$.
       - 1a2b. Do 1 iteration of approximate solve: $A_i u_i = f_i$.
       - 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: $A_i u_i = f_i$.

2a3. End Do

2b. End Do

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	
		Ex. 2	Ex. 3
Approx. Solve	A_i, u_i, f_i	$(4, 5)$	$(1, 5)$
Residual	A_i, u_i, f_i, r_i	$(5, 5)$	$(1, 5)$
Restrict.	r_i, f_{i+1}	$(.5, 2)$	$(.5, 2)$
Prolong.	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. On a serial computer, this is just a multiplicative domain decomposition algorithm. On a parallel computer, this can be an additive 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 reduction in cache hits per type of data over the usual implementations is given by the following:

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	–

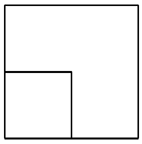
Recalling that m_i is typically 2 or 4, these reductions are quite significant.

Algorithm DDM-Cache-Vcycle

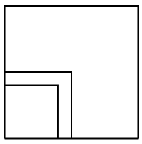
(k, $\{A_i, u_i, f_i, r_i\}_{i=1}^{k}$)

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

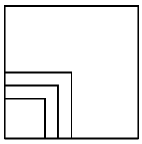
5. One Pass Through Cache



Compute on Ω_{ij}: one iteration

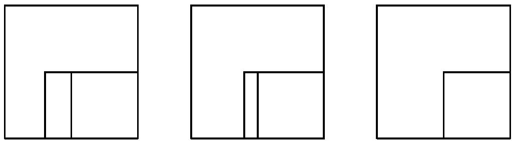


Compute on subset of Ω_{ij}: second iteration with 1 line buffer



For each iteration, add 1 more pair of lines to buffer

While computing in neighboring Ω_{ij}'s, the rest of the updates are done carefully.



V Cycle becomes:

Compute
V Cycle on coarse grid
Compute

where everything is done in a complicated subroutine.

Great, but... coding this is a miserable task.

6. Automatic Cache Detection

[joint with O. J. Anshus, J. Edlar, K. Li, and J. Philbin]

We want an inexpensive software library to determine Ω_{ij}. Such a tool exists in a simple form.

th_init($cdots$)
th_fork(function, i, j, address1, address2)
th_run($cdots$)
th_stats()

Hints provided to a "very light weight threads" library. A hashing function maps into a 2D cache table. The th_run goes through the hash table "buckets" and calls the given function. For iterative methods, the order changes usually. Library is ~500 lines of C.

In my examples so far, using this library to make standard codes cache aware leads to run times that are ~5% slower than hand coded schemes, at least for simple cache aware schemes.

This library is still too simple for unstructured grid multigrid. An effort is under way to deal with this. While the locality of data used at a point is nice, needing to know where all of the data is should not have to be required. This is supposed to be a *fast*, approximmate cache determination library, not an optimal one.

If this works, then there is a connection to load balancing for parallel multigrid (or linear algebra). For PDE problems, load balancing means that the computation and communication is similar enough for all processors. The threads library does this for caches. A cache, in theory, is what a local memory on a distributed memory parallel computer resembles. So, "super elements" can be determined, one or more per processor. (At least in theory.)

7. 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 here 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 or additive domain decomposition method saves even more time and allows us to use theoretical convergence rates from that field for our problems.

Implications of Work

- Parallel computer design
  Spend more cache, less on standard RAM.

- Good/bad cache designs
  More complicated cache aware multigrid algorithms needed for bad cache designs.

- Automatic cache determination
  A modest overhead is acceptable when programming is so complicated instead.

- Examples
  Will be on MGNet, summer 1996.

- Spaghetti multigrid
  Requires a highly structured implementation.

8. References

1. C. C. Douglas, Caching in with multigrid algorithms: problems in two dimensions, IBM Research Report RC 20091, Yorktown Heights, NY, 1995 and CERFACS Report TR/PA/95–15, Toulouse, France, 1995.

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

3. J. Philbin, J. Edlar, O. J. Anshus, C. C. Douglas, K. Li, Thread Scheduling for Cache Locality, Preprint, 1996.

Good URL's (WWW):
http://www.cerfacs.fr/~douglas/
http://www.cs.yale.edu/users/douglas-craig/

Good anonymous ftp sites:
ftp.cerfacs.fr and *casper.cs.yale.edu*