THE STATE OF QUASI-UNSTRUCTURED GRID CACHE AWARE MULTIGRID*

CRAIG C. DOUGLAS†, GUNDOLF HAASE‡, AND JONATHAN HU§

Abstract. The numerical solution of PDEs on unstructured grids provides a challenge for high performance computing. Quasi-unstructured grids are typically embedded in unstructured grids. For a trivial preprocessing cost (particularly for time dependent problems), many unstructured grid problems can be converted into a structured grid case with a tiny truly unstructured subdomain.

In this paper, algorithms that reuse caches efficiently are examined for several classes of problems. The algorithms provide bitwise exactly the same solutions as standard non-cache aware algorithms. Hence, the convergence rates are identical. However, the cache aware algorithms run significantly faster than the standard algorithms.

1. Introduction. In recent years, computer processors have vastly outstripped advances in memory chips' abilities to read or write data. Superior use of memory hierarchies, where each level's memory is referred to as a cache, leads to dramatically faster codes.

Cache aware solvers are designed to use cache memory in ways that compilers are incapable of doing due to crucial information about data that is only available at runtime. Cache aware solvers for (coupled) partial differential equations (PDE's) are now absolutely essential to achieving anything other than poor performance when measured against possible peak performance figures. We describe a set of techniques to achieve a respectable percentage of peak performance.

For decades algorithm complexity has been measured by the number of floating point operations. Cache memory hierarchies can make this methodology misleading for the same reason that a more complex algorithm can be faster on a newer computer architecture than a simpler algorithm on an older architecture. We suggest an alternative measure: the number of expected cache misses, where a miss is defined as any memory reference that causes data to be moved further up in the cache memory hierarchy.

Figure 1.1 gives relevant ratios of the cost of moving data around in a computer. Main memory is divided in this table to represent a trend that is becoming more common on shared memory nodes of a symmetric multiprocessor node. On a PC or workstation, main memory can be combined into a single entry.

In this paper we show that the preprocessing costs are not as great as first expected. We show that, for a collection problems, the state of cache aware multigrid is now ready for user friendly libraries.

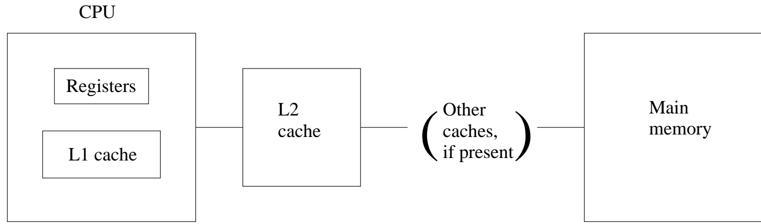
In §2, we define the algorithms. In §3, we analyze the cost of constructing cache aware algorithms. In §4, we compare actual run times with the theory for some examples. In §5, we draw conclusions.

*This research has been partially supported by the NSF (grants DMS-9707040, ACR-9721388, ACR-9814651, CCR-9902022, and CCR-9988165), Sandia National Laboratory, and gifts from the Intel and Hewlett-Packard Corporations.

†University of Kentucky, Department of Computer Science, 325 McVey Hall-CCS, Lexington, KY 40506-0045, USA. douglas@ccs.uky.edu. Also, Yale University, Department of Computer Science, P.O. Box 208285, New Haven, CT 06520-8285, USA. douglas-craig@cs.yale.edu.

‡Johannes Kepler University of Linz, Institute for Computational Mathematics, Altenberger Str. 69, A–4040 Linz, Austria. ghaase@numa.uni-linz.ac.at.

§Sandia National Laboratories, Livermore, CA 94551-0969, USA. jhu@california.sandia.gov.



(a) Memory hierarchy.

Level	clocks
register	1
L1 cache	2–3
L2 cache	6–12
near main memory	60–100
far main memory	100–250
distributed main memory	$\mathcal{O}(100)$
message-passing	$\mathcal{O}(1000)$–$\mathcal{O}(10000)$

(b) Clock speeds.

FIGURE 1.1. *Memory hierarchy and speeds*

2. Algorithms. In this section we define the algorithms that we use. This includes multigrid and cache aware stationary iterative methods.

We assume that we have a sequence $j = 1, \ldots, L$ of solution spaces $\mathcal{M}_j$ such that $\mathcal{M}_j \subset \mathcal{M}_{j+1}$, discrete problems $\mathcal{A}_j u_j = f_j$, solvers $\mathcal{S}_j : \mathcal{M}_j \to \mathcal{M}_j$, prolongation operators $\mathcal{P}_j : \mathcal{M}_j \to \mathcal{M}_{j+1}$, and restriction operators $\mathcal{R}_j : \mathcal{M}_j \to \mathcal{M}_{j-1}$.

The two most common multigrid algorithms are the V and W cycles (see Fig. 2.1). The solvers referred to in the figures are typically scaled iterative methods with Gauss-Seidel as the most common. On the coarsest grid, a direct solver may be substituted for the iterative solver. The solvers are usually smoothers [3], but may also be roughers [5, 6]. A smoother is an iterative method (e.g., Gauss-Seidel) such that the magnitudes of error components are guaranteed not to be increased. A rougher is any iterative method where the magnitude of an error component may be increased (e.g., a Krylov subspace method or SSOR [2, 5, 6]).

On a level we do one of four operations depending on where in the cycle we are. Assume that there is an initial guess u_j^0 before the smooth/rough step.

- Going from a fine grid to a coarser grid:
  - Smooth/rough: For $i = 1, \ldots, m$, $u_j^i \leftarrow \mathcal{S}_j(u_j^0, f_j)$.
  - Set $u_j \leftarrow u_j^m$, $f_{j-1} \leftarrow \mathcal{R}_j(f_j - \mathcal{A}_j u_j)$, and $u_{j-1}^0 \leftarrow 0$.

  Note that the residual $f_j - \mathcal{A}_j u_j$ can easily be computed during the last iteration of the iterative procedure. Usually this results in significant reuse of data in cache.
- Going from a coarse grid to a finer grid:
  - Set $u_j^0 \leftarrow u_j + \mathcal{P}_{j-1} u_{j-1}$.
  - Smooth/rough: For $i = 1, \ldots, m$, $u_j^i \leftarrow \mathcal{S}_j(u_j^0, f_j)$.
  - Set $u_j \leftarrow u_j^m$.

  Projection can be combined with the last iteration, but usually is not.
- On the coarsest grid:
  - Solve $\mathcal{A}_1 u_1 = f_1$.

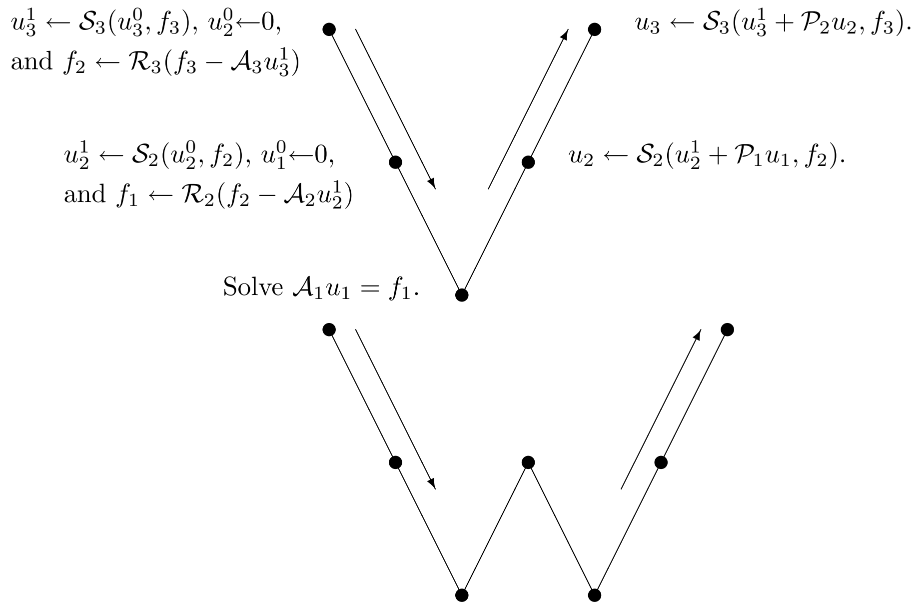


FIGURE 2.1. *V and W cycles for solving* $\mathcal{A}_3 u_3 = f_3$.

Interpolation can be combined with the first iteration, but usually is not.

- On the finest grid at the end of a cycle:
  - Smooth/rough: For $i = 1, \ldots, m$, $u_j^i \leftarrow \mathcal{S}_j(u_j^0, f_j)$.
  - Set $u_j \leftarrow u_j^m$.

  A final residual calculation may or may not be needed. If it is, it can be calculated during the last iteration with a significant cache reuse.

Our experience has shown that about 80% of the calculation time is spent on the iterative solver and residual calculation for realistic parameter choices (e.g., number of iterations of the solver and multigrid cycles) [7]. While a completely cache aware multigrid approach has been done in a simple case [4], the extra effort of incorporating either the restriction or the interpolation into the last step of the smoother/rougher is not worth either the aggravation or the added code complexity.

We motivate our work with a simple example. Consider the small grid (37 nodes) in Fig. 2.2. Now suppose that data for only 18 nodes of the problem fits into cache. A standard implementation of Gauss-Seidel updates nodes in order of increasing node number. This leads to poor use of cache: by the time node 37 is updated, information for node 1 has been evicted from cache. Hence, data for each unknown must be brought into cache during every iteration.

There are two alternative approaches:

- *Active set approach*: A fixed size subdomain slides across the domain during computation.
- *Explicit cache blocking*: The mesh and corresponding matrix operator A_j are decomposed into contiguous, nonoverlapping subdomains, where the subdomain size depends on the size of the cache. This approach is similar to a domain decomposition method.

Each method is complicated since the iterates are required to be the same whether a traditionally coded method or one of the two cache aware methods are used.

First consider an active set approach. A data set is formed using an initial subset

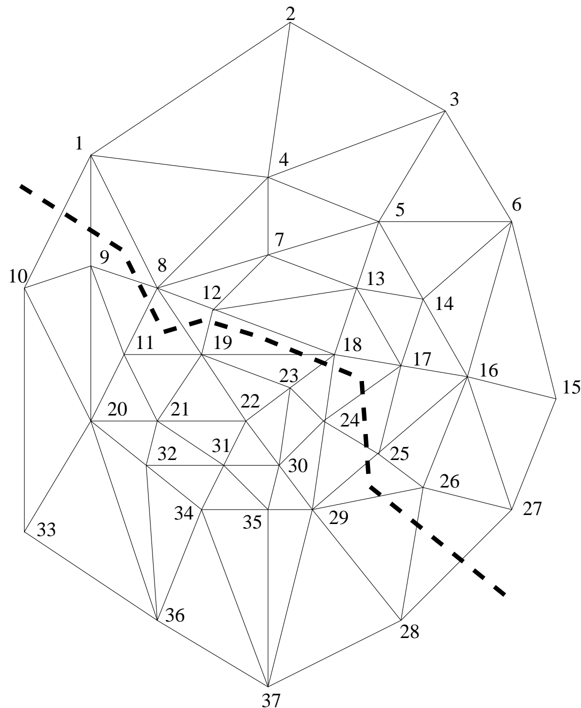


Figure 2.2. *Motivating example grid*

of the vertices in the grid. All of the unknowns associated with the active set are updated once. Iteratively, a second, third, ... time as many of the unknowns are updated as is possible and still keep the bitwise compatibility that we require. Once an unknown is updated the number of times it needs to be, then it is removed from the active set and a new vertex is added. The active set needs to be large enough so that all of the associated data fits into cache simultaneously.

Now consider the approach of explicitly blocking for cache. We again use the grid in Fig. 2.2 to illustrate this approach. We assume that the cache is sufficiently large to hold all data from one multigrid level for about half of the nodes but not large enough to hold the data for all of the nodes. The example grid is divided into two connected subdomains (see Fig. 2.3). Then renumber the grid points inside each subdomain so that nodes that are further way from the other subdomain have a lower index than closer nodes. We can then update as much as possible within one subset before visiting the other. This approach leads to superior data reuse within the cache.

Suppose we want to do three updates per node. The circles around nodes in Fig. 2.3 indicate how many updates can be done per node without interacting with the other subdomain. Had there been thousands of nodes per subdomain (instead of 17 or 18), the vast majority would normally be updated three times without interacting with another subdomain.

Note that some unknowns can be completely updated if the subdomains are big enough. In addition most of the residuals can be calculated at the completed points.

For the points nearest the boundary between the subdomains, their updates cannot be completed in one pass through the cache subdomain. However, only the incomplete points have to be revisited later. In a realistic situation, the subdomains are

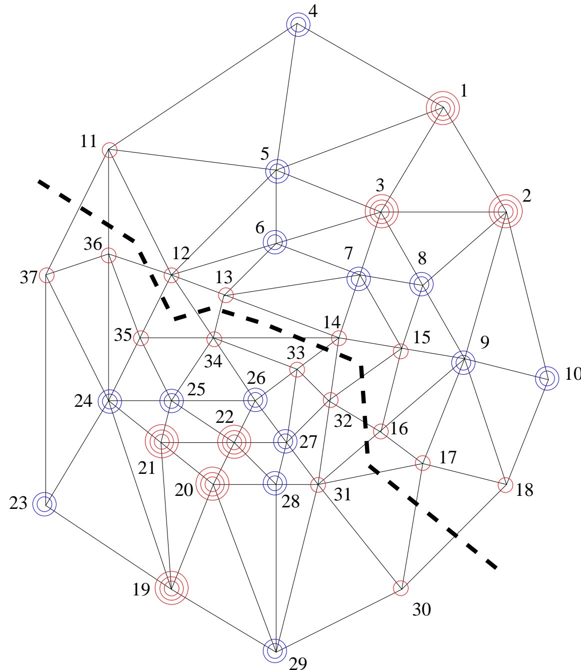


FIGURE 2.3. *Two subdomain decomposition of example grid*

large enough so that the vast majority of points in a subdomain can be completely updated without a revisit.

Designing an efficient and portable cache aware Gauss-Seidel algorithm requires a preprocessing step in addition to a new way of running the basic algorithm. Based on a new ordering we want the standard and cache aware implementations to get bitwise identical results. Hence, the norm of the difference of the two implementations is zero unless one of the codes has an error in it.

By portable, we mean that we minimize the number of parameters. We use only half of the cache in the closest cache to the registers that is large enough to be useful [9]. Half was determined by experiments that tried to determine how much of cache remains intact on many multitasking platforms, where other tasks interfere with our computations.

In the preprocessing step we perform three steps. First we decompose each mesh into disjoint cache blocks, e.g., [8]. Then we renumber the nodes in each block [1]. Finally, we produce the system matrices and intergrid transfer operators with the new ordering.

The Gauss-Seidel iteration also has three steps. First we update as much as possible within a block without referencing data from another block. Then we calculate a (partial) residual on the last update. Finally, we backtrack to finish updating cache block boundaries.

In the mesh decomposition preprocessing step we have goals, constraints, and a critical parameter. Our goal is to maximize the interior of each cache block while minimizing the connections between cache blocks. The optimal solution to our goal is too expensive. Hence, we aim for something less than optimal, but good enough to be effective.

We have the constraint that the cache should be large enough to hold those parts of the matrix, right hand side, residual, and unknown that are associated with a cache block. The critical parameter is the cache size. This problem has been studied in depth for load balancing on parallel computers.

3. Algorithms for Renumbering Nodes. A critical component of the Gauss-Seidel iteration is a mesh renumbering step that potentially could overwhelm (in time) the benefits of the cache aware algorithm. The preprocessing step in this optimization requires a mesh decomposition into connected cache blocks and renumbering the nodes within each block. The choice of renumbering algorithms depends on whether certain connectivity information is available.

3.1. Definitions. Assume that the physical domain $\Omega \subset \mathbb{R}^2$ is simply connected, i.e., Ω cannot be made disconnected by the removal of a finite number of points. We consider only decompositions of the discretized domain such that the subdomains are simply connected. Furthermore, for each subdomain Ω_i and Ω_j, the curve described by the boundary (sequence of nodes and edges) of Ω_i does not surround Ω_j.

We first give some basic graph definitions. Let $\mathcal{G}$ be the graph with nodes

$$\mathcal{N}(\mathcal{G}) = \{u : u \text{ is a node in grid}\}$$

and edges

$$\mathcal{E}(\mathcal{G}) = \{(u, v) : u \in \text{nbr}(v)\}.$$

When no confusion can arise, we write

$$\mathcal{N} := \mathcal{N}(\mathcal{G}) \text{ and } \mathcal{E} := \mathcal{E}(\mathcal{G}).$$

A *subgraph* $\mathcal{G}'$ of $\mathcal{G}$ is any graph with nodes $\mathcal{N}' \subset \mathcal{N}$ and edges $\mathcal{E}' \subset \mathcal{E}$ such that any edge of $\mathcal{G}'$ has both endpoints in $\mathcal{N}'$.

A *path* P between nodes i and j is a set of edges $(i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k)$, where $i_1 = i$ and $i_k = j$. The *length* of P is $|P| = k + 1$. This is a nonstandard definition and is made for convenience of indexing in later sections. For example, the distance from any node to itself is 1. Any subgraph $\mathcal{G}'$ is *connected* if for any pair of nodes i, j in $\mathcal{G}'$, there exists a path between i and j such that all nodes in the path are also in $\mathcal{G}'$. A *component* of a graph $\mathcal{G}$ is a maximally connected subgraph.

We now define the different sets of nodes that are used in the renumbering algorithms and complexity theorems. The grid in Figure 2.3 is used to illustrate these definitions. A node on the physical boundary of Ω is called a *physical boundary node*. For example, nodes $1, 2, 4, 10, 11, 18, 19, 23, 29, 30,$ and 37 are the physical boundary nodes. Any node that is not a physical boundary node is a *physical interior node*. Define the set of neighbors of a node u to be $\text{nbr}(u) = \{v \in \mathcal{T} : \text{the discrete solution at } u \text{ depends on the solution at } v\}$. For example, $\text{nbr}(3) = \{1, 2, 5, 6, 7, 8\}$ if we assume that the graph connections reflect the matrix stencil. Define a *cache boundary node* to be any node i such that i has a neighbor j that is in a different cache block. For a given cache block, the set of all cache boundary nodes is called the *cache boundary*. For the lower left cache block, nodes 30–37 are cache boundary nodes. For a given subgraph, any member node that is not a cache boundary node is a *cache interior node*. Recall that nodes 19, 23, 29, 30, and 37 are physical boundary nodes. All other nodes in the lower left block are cache interior nodes.

Table 3.1

Quantities used in the complexity analysis of the cache Gauss-Seidel preprocessing phase.

N_Ω	Number of nodes in cache block Ω
K	Average number of connections per node
d	Number of degrees of freedom per node

The *distance* D_v of a member node v from the cache block boundary is $\min\{|P| : P$ is a path from v to a boundary node$\}$. The minimum length path from any node to itself contains zero edges. Therefore the distance of a cache boundary node to the cache boundary is one. For any subgraph $\mathcal{S}$, define *subblock* $\mathcal{S}_i$ to be the set $\{v \in \mathcal{N}(\mathcal{S}_i) : D_v = i\}$. For example, nodes 30–37 form subblock $\mathcal{S}_1$ of the lower left cache block and nodes 23–29 form subblock $\mathcal{S}_2$.

The blocks may be produced after using a load balancer for parallel computing or a graph decomposition system such as METIS [8]. Cache blocks and distributed memory domain decomposition strategies have much in common. Hence, we can use the extensive research and code development from these fields.

3.2. Complexity of Standard Gauss-Seidel. Table 3.1 gives definitions of the constants used in the complexity analysis. Note that all nodes have d degrees of freedom. Some of these degrees of freedom may have prescribed values, i.e., Dirichlet boundary conditions. It is assumed, however, that in practice these unknowns are still present in the linear systems. Note that a node i with K_i connections is a vertex of $K_i - 1$ distinct elements if i is a physical boundary node. Otherwise i is a vertex of K_i triangular elements.

Theorem 3.1. *The complexity $\mathcal{C}_{gs}$ of a standard Gauss-Seidel sweep over N nodes with at most d degrees of freedom per node is given by*

$$\mathcal{C}_{gs} = 2d^2 N_\Omega K + d.$$

3.3. Cache boundaries are connected. Consider a cache block decomposition in which all cache boundaries are connected. Algs. 2 and 1 mark all nodes in a cache block.

Alg. 2 assumes that all cache boundaries are connected. An initial cache boundary node is found and pushed onto stack S_1. A node i is removed from S_1 and scanned. If i is a cache boundary node, then i is marked as distance one, and all unmarked adjacent nodes j that are in Ω_s are marked as distance -1 (so that j will not be pushed onto S_1 more than once) and pushed onto S_1 for later scanning (lines 7-12). Otherwise, i is marked as distance two from the cache boundary and pushed onto stack S_2 (lines 13-15). This process repeats until S_1 is empty.

The contents of S_2 are now moved to S_1. Alg. 1 removes each node i from S_1 and scans it (lines 8-13). Each unmarked neighbor j of i is marked and pushed onto S_2. Once S_1 is empty, the stacks switch roles. Once $m+1$ subblocks have been identified, all remaining unmarked nodes are marked as distance $m+1$ from the cache boundary. Alg. 1 continues until both stacks are empty. At the conclusion of Alg. 1, vector D contains the smaller of $m+1$ and the minimum distance from each node of block Ω_s to the boundary $\partial\Omega_s$.

Theorem 3.2. *The complexity $\mathcal{C}_1$ of Algorithm 1 is bounded by*

$$\mathcal{C}_1 \le 5 N_\Omega K.$$

Algorithm 1 Mark cache interior nodes.

Label-Internal-Nodes

```
 1: Set current distance c = 3.
 2: Let m be the number of subblocks desired.
 3: while S_2 is not empty do
 4:    Move contents of S_2 to S_1.
 5:    while S_1 is not empty do
 6:       Pop node i off S_1.
 7:       for each node j adjacent to i do
 8:          if distance D_j == 0 then
 9:             Set distance D_j = c.
10:             Push j onto S_2.
11:          end if
12:       end for
13:    end while
14:    if c < m then
15:       Set c = c + 1.
16:    end if
17: end while
```

Algorithm 2 (cache boundaries connected) Mark cache boundary nodes.

Label-Boundary-Nodes

```
 1: Initialize stacks S_1 and S_2.
 2: Set D_i = 0 for all i in cache block Ω_s.
 3: Find any node i on cache boundary ∂Ω_s.
 4: Push i onto S_1.
 5: while S_1 is not empty do
 6:    Pop node i off S_1.
 7:    if i is in ∂Ω_s then
 8:       Set D_i = 1.
 9:       for each node j connected to i do
10:          if D_j == 0 and j is in Ω_s then
11:             Set D_j = −1.
12:             Push j onto S_1.
13:          end if
14:       end for
15:    else
16:       Set D_i = 2.
17:       Push i onto S_2.
18:    end if
19: end while
```

The complexity $\mathcal{C}_2$ *of Algorithm 2 is bounded by*

$$\mathcal{C}_2 \leq (7K+1)N_\Omega.$$

The cost of Algs. 1 and 2 on the finest grid is at most $6d^{-2}$ *sweeps of standard Gauss-Seidel on the finest grid.*

We note that the estimate in Theorem 3.2 is pessimistic. In the proof of 3.2, certain values that are much smaller than N_{Ω_s} are bounded by N_{Ω_s} for convenience.

3.4. Physical boundary nodes are unknown. Now consider the case where we do not know which nodes lie on the physical boundary. We again present two algo-

rithms for renumbering the nodes and analyze their complexity in terms of standard Gauss-Seidel sweeps.

Algorithm 3 (unknown physical boundary nodes) Mark cache boundary nodes.

Label-Boundary-Nodes

1: Initialize stacks S_1 and S_2.
2: Set distance $D_i = 0$ for all i in Ω_s.
3: **for** each node i in Ω_s such that $D_i == 0$ **do**
4:   **if** i is on $\partial\Omega_s$ **then**
5:     Push i onto S_1.
6:     **while** S_1 is not empty **do**
7:       Pop node i off S_1.
8:       **if** i is in $\partial\Omega_s$ **then**
9:         Set $D_i = 1$.
10:         **for** each node j connected to i **do**
11:           **if** $(D_j == 0)$ AND (j is in Ω_s) **then**
12:             Set $D_j = -1$.
13:             Push j onto S_1.
14:           **end if**
15:         **end for**
16:       **else**
17:         Set $D_i = 2$.
18:         Push i onto S_2.
19:       **end if**
20:     **end while**
21:   **end if**
22: **end for**

Alg. 3 finds distances of all cache boundary nodes. Alg. 1 again finds the distances of all cache interior nodes. Alg. 3 examines each node in a cache block until an unmarked boundary node is found. This node is pushed onto S_1 (lines 4-5). The WHILE loop (lines 6-23) is identical to that in Alg. 2 (see §3.3.) Each time S_1 is emptied, the FOR loop (line 3) iterates until an unmarked cache boundary node is found. The algorithm finishes when S_1 is empty and there are no more unmarked cache boundary nodes. At the conclusion of Alg. 3, all cache boundary nodes have been marked as distance one, and all nodes that are distance two from the boundary have been marked and placed on S_2 in preparation for Alg. 1.

THEOREM 3.3. *The complexity $\mathcal{C}_3$ of Algorithm 3 is bounded by*

$$\mathcal{C}_3 \leq 9 N_\Omega K.$$

The cost of Algs. 1 and 3 is at most $7d^{-2}$ sweeps of standard Gauss-Seidel on the finest grid.

This estimate is pessimistic. We also note that while it appears that Alg. 2 is embedded in Alg. 3, the WHILE loop (lines 6-16) in Alg. 3 is initiated only when a new cache boundary component is found. The WHILE loop thus iterates until the component is fully marked.

3.5. Physical boundary nodes are known. Now assume that we know which mesh nodes are physical boundary nodes. If one cache block boundary node is known, we can find all cache block boundary nodes by following paths consisting of physical boundary nodes that are in the current cache block. This is the motivation for Algs. 4 and 5.

First consider Alg. 4. Stack S_4 is the set of physical boundary nodes that are in unexplored cache boundary components. S_4 is searched until an unmarked node is found. This is the starting point of the WHILE loop (lines 12-24). This WHILE loop repeats until all nodes in the cache boundary component marked. Each cache boundary node in S_1 is marked as distance one (line 15), and any neighbor that is unmarked and in the same component is marked and pushed onto S_1 (lines 17-19). Every cache interior node in S_1 is marked as distance two and pushed onto S_2 (lines 21-22). If a node of distance two is also a physical boundary node, it is also pushed onto stack S_3 (lines 21-25). Nodes on S_3 are potential starting points of physical boundary node paths connecting the current component to other components in the cache block boundary.

Once S_1 is empty and cannot be renewed from S_4, Alg. 5 grows paths from the nodes in S_3 until another unexplored component is found. The algorithms stop when S_4 cannot be refilled.

Algorithm 4 (known physical boundary nodes) Main algorithm for marking cache boundary nodes.

Label-Boundary-Nodes

1: Let PBN denote a physical boundary node.
2: Initialize stacks S_1, S_2, S_3, S_4.
3: Set $D_i = 0$ for each node i in Ω_s.
4: Find any node k in $\partial\Omega_s$, and push k onto stack S_4.
5: **repeat**
6:   **repeat**
7:     **while** ($S_1 == \emptyset$) AND ($S_4 \neq \emptyset$) **do**
8:       Pop k off stack S_4.
9:       **if** $D_k == 0$ **then**
10:         Set $D_k = 1$.
11:         Push k onto S_1.
12:       **end if**
13:     **end while**
14:     **while** $S_1 \neq \emptyset$ **do**
15:       Pop i off of S_1.
16:       **if** i is in $\partial\Omega_s$ **then**
17:         Set $D_i = 1$.
18:         **for** each node j connected to i **do**
19:           **if** ($D_j == 0$) AND (j is in Ω_s) **then**
20:             Set $D_j = -1$.
21:             Push j onto S_1.
22:           **end if**
23:         **end for**
24:       **else**
25:         Set $D_i = 2$.
26:         Push i onto S_2.
27:         **if** i is a PBN **then**
28:           Push i onto S_3.
29:         **end if**
30:       **end if**
31:     **end while**
32:   **until** S_4 is empty.
33:   Call Find-Next-Cache-Boundary-Component(S_3, S_4).
34: **until** S_4 is empty

Algorithm 5 (known physical boundary nodes) Finds next cache boundary component for Alg. 4.

Find-Next-Cache-Boundary-Component(S_3, S_4)

```
 1: while S_3 ≠ ∅ do
 2:   Pop i off of S_3.
 3:   if i is in ∂Ω_s then
 4:     Push i onto S_4.
 5:   end if
 6:   for each node j connected to i do
 7:     if (j ∈ Ω_s) AND (D_j == 0) AND (j is a PBN) then
 8:       Set D_j = −1.
 9:       Push j onto S_3.
10:     end if
11:   end for
12: end while
```

THEOREM 3.4. *The complexity $\mathcal{C}_{1,4,5}$ of Algorithms 1, 4, and 5 is bounded by*

$$\mathcal{C}_{1,4,5} \leq (20K + 2)N_\Omega.$$

The cost of Algs. 1, 4, and is at most $10d^{-2}$ sweeps of standard Gauss-Seidel on the finest grid.

In Theorem 3.4 we have assumed that the number of physical boundary nodes in a cache block is $\mathcal{O}(N_{\Omega_s}^{\frac{1}{2}})$. This estimate is pessimistic.

Consider what happens to the bound if we assume that at most half of any cache block is physical boundary nodes.

THEOREM 3.5. *Let the number of physical boundary nodes be no more than half of N_Ω. Then the complexity $\mathcal{C}_{1,4,5}$ of Algorithms 1, 4, and 5 is bounded by*

$$\mathcal{C}_{1,4,5} \leq (11K + 1)N_\Omega.$$

Then the cost of Algs. 1, 4, and 5 is at most $5.5d^{-2}$ sweeps of standard Gauss-Seidel on the finest grid.

4. Numerical Results.

In this section we compare actual run times of five level V cycles on examples to see how the state of cache aware multigrid matches theory. Our examples consist of an elasticity problem on a domain shaped like Austria (see §4.1) and a stationary heat equation on domains shaped like Bavaria or Kentucky (see §4.2).

Our results are on two platforms: a 64 bit Itanium (IA-64) processor and a 32 bit Pentium III (IA-32) processor. Specifics for each platform are in Table 4.1. The datasets are too large to fit into the L2 caches on either platform.

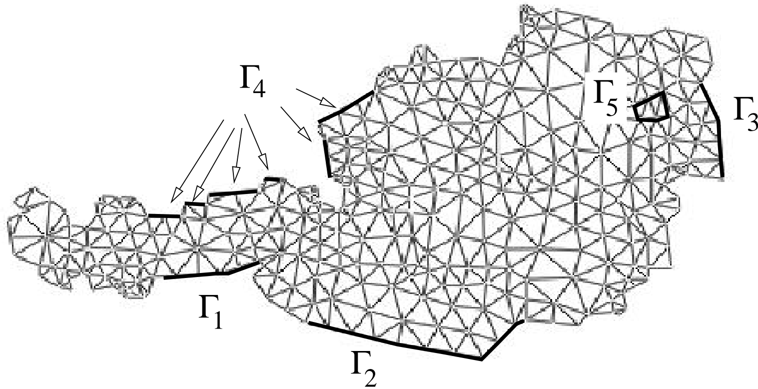
4.1. Two degrees of freedom problem.

The first test problem is solving a two dimensional linear elasticity problem on a domain in the shape of Austria (see Fig. 4.1(a). We have two degrees of freedom per vertex.

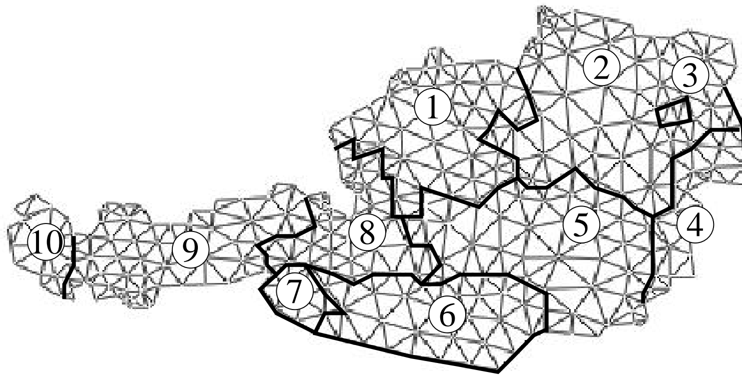
The domain is divided into 10 regions, each consisting of an isotropic material. Material properties may vary from region to region, which leads to internal material discontinuities at some region boundaries. See Table 4.2 for the material properties. See Fig. 4.1(b) for the corresponding regions. The continuous problem is a two

Table 4.1
Computing platforms used in numerical experiments.

	Intel Itanium (IA-64)	Intel Pentium III (IA-32)
CPU	733 MHz	500 MHz
OS	Linux	Linux
L1 cache	32 byte lines 4 way associative 16+16 KB on chip	32 byte lines 8 way associative 16+16 KB on chip
L2 cache	32 byte lines 6 way associative 96 KB on chip	32 byte lines 8 way associative 256 KB off chip
L3 cache	16 byte lines 2.00 MB	None None



(a) Coarsest grid with boundaries marked



(b) Regions of different materials. See Table 4.2 for the material properties.

Figure 4.1. *Elasticity problem domain definition*

Table 4.2
Material properties for Equation (4.1) defined on Austria domain.

Region	1	2	3	4	5	6	7	8	9	10
Young's modulus	1	1	10000	1	1	10	1	1	1	1

TABLE 4.3
Number of mesh nodes and unknowns for each multigrid level in Austria example.

Multigrid level	1	2	3	4	5
Mesh nodes	387	1210	4567	17725	69817
Unknowns	674	2420	9134	35450	139634

dimensional second order self adjoint elliptic PDE. The governing equation is

$$\begin{cases} -\nabla T = f, & \text{in } \Omega, \\ \frac{\partial w}{\partial n} = 100w, & \text{on } \Gamma_1, \\ \frac{\partial w}{\partial y} = 100w, & \text{on } \Gamma_2, \\ \frac{\partial w}{\partial x} = 100w, & \text{on } \Gamma_3, \\ \frac{\partial w}{\partial n} = 0, & \text{everywhere else.} \end{cases} \tag{4.1}$$

where T is the Cauchy stress tensor, w is the displacement, and f is the forcing vector given by

$$f = \begin{cases} (9.5 - x, 4 - y), & \text{if } (x, y) \text{ is in region surrounded by } \Gamma_5 \\ (1, -1)^{\mathrm{T}}, & \text{on } \Gamma_4 \\ 0, & \text{otherwise.} \end{cases}$$

T is given by

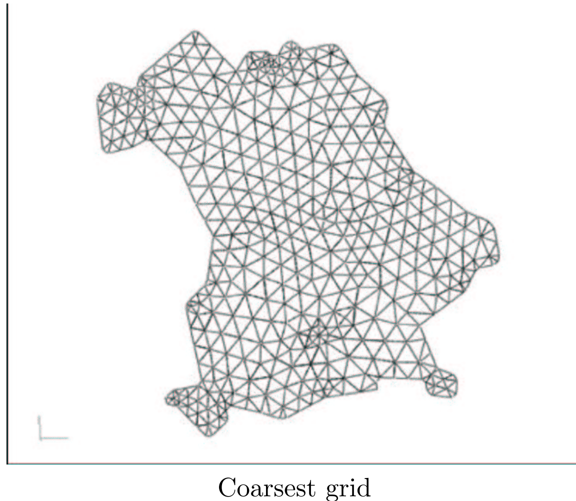
$$T = C[E]\ 2\mu E + \lambda(\mathrm{tr}E)I$$

where $E[w] = 0.5(\nabla w + (\nabla w)^{\mathrm{T}})$ is the strain tensor, C is the elasticity tensor, and λ and μ are the Lamé constants. The domain is discretized with linear triangular elements, and each mesh node has two degrees of freedom. See Table 4.3 for the number of mesh nodes and unknowns per level. The problem is discretized into five multigrid levels. The coarse level matrix is solved directly.

The speedups achieved over a standard implementation are given in Figs. 4.3 and 4.4. The active set approach is 10-25% and 33-50% slower for the IA-64 and IA-32 processors, respectively, than the cache blocking approach in this example. Table 4.4 shows that the preprocessing time is about 1.1 and 1.4 times a standard Gauss-Seidel sweeps on the IA-64 and IA-32 platforms, respectively.

4.2. One degree of freedom problems. The second test problem is a two dimensional stationary heat equation on a domain shaped like Bavaria (see Fig. 4.2) with 7 sources and one sink (corresponding to Munich). The boundary conditions are homogeneous Dirichlet on the Czech border (along the northeast) and homogeneous Neumann elsewhere. Another stationary heat problem on a domain in the shape of Kentucky is also included in the speedup summary. There is only one degree of freedom per vertex.

The speedups achieved over a standard implementation are given in Figs. 4.3 and 4.4. The active set approach and the cache blocking approaches are equivalent in both the Bavaria and the Kentucky examples. Table 4.4 shows that the preprocessing time is between 1.4 and 1.5 standard Gauss-Seidel sweeps.



Coarsest grid

Figure 4.2. *Heat equation Bavarian domain definition*

Table 4.4
Ratio of preprocessing time to one Gauss-Seidel sweep

Platform	Austria	Bavaria	Kentucky
IA-32	1.4	1.5	1.4
IA-64	1.1	1.5	1.5

5. Conclusions. In problems where unstructured grids are reused many times, cache aware multigrid provides very good speedups. Speedups of 20-100% are significant for problems requiring large amounts of CPU time.

Two blocking methods were investigated. The *explicit cache blocking* method is consistently as fast as the *active set* and can be as much as 25% faster in our examples. The active set speedup is particularly disappointing when there are multiple degrees of freedom, whereas the explicit cache blocking method does much better.

While we do not get the last possible floating point operation per second from a CPU, we get a respectable percentage of the published peak rate. Our implementation is not tuned for a particular architecture meaning that it is highly portable. In particular, the available cache size is the only tuning parameter.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, MA, 1974.

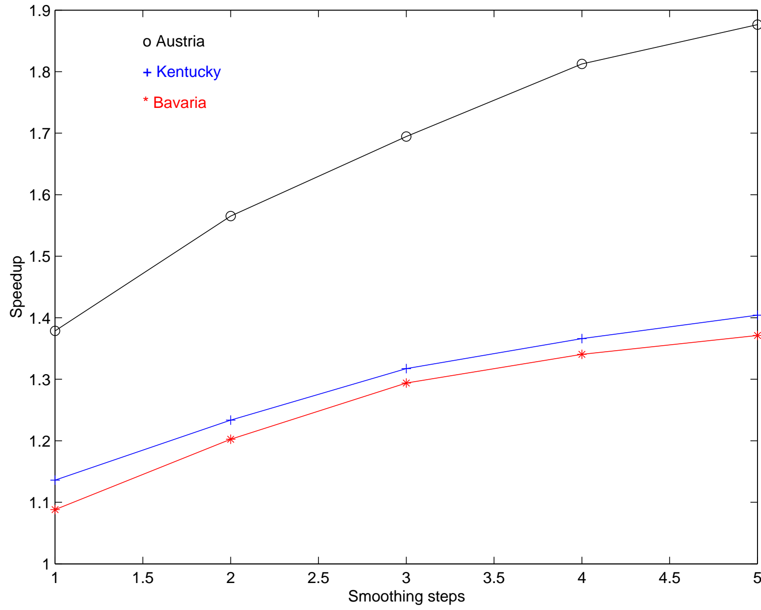
[2] R. E. Bank and C. C. Douglas. Sharp estimates for multigrid rates of convergence with general smoothing and acceleration. *SIAM J. Numer. Anal.*, 22:617–633, 1985.

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

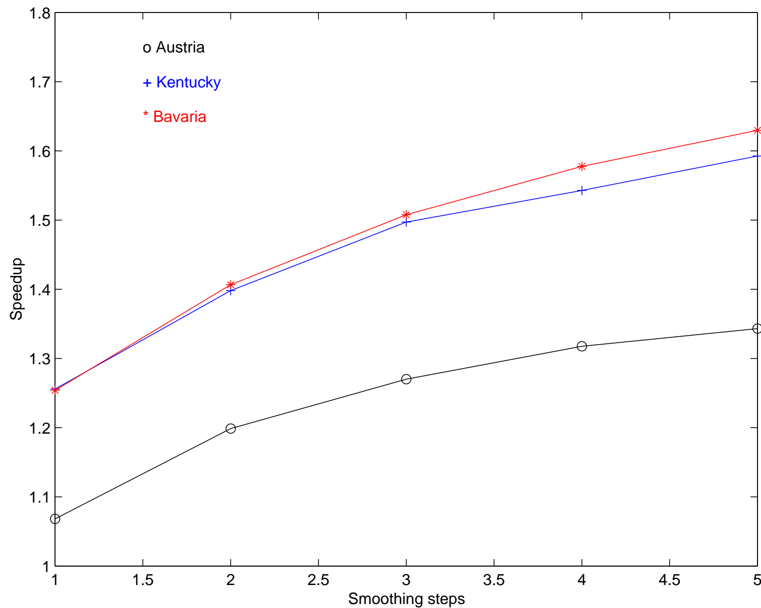
[4] C. C. Douglas. Caching in with multigrid algorithms: problems in two dimensions. *Paral. Alg. Appl.*, 9:195–204, 1996.

[5] 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.

[6] 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.



(a) Explicit cache blocking speedups



(b) Active set speedups

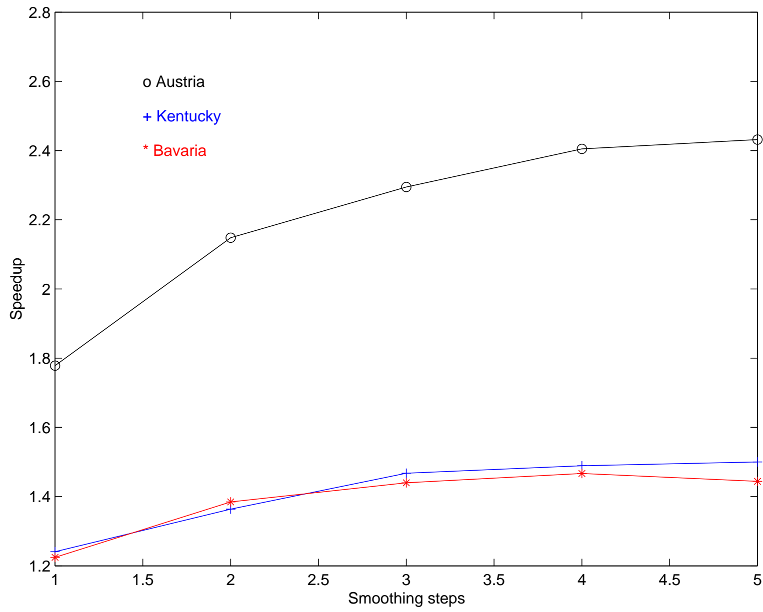
FIGURE 4.3. *Speedups on Linux based Itaniums (IA-64)*

[7] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Elect. Trans. Numer. Anal.*, 10:21–40, 2000.

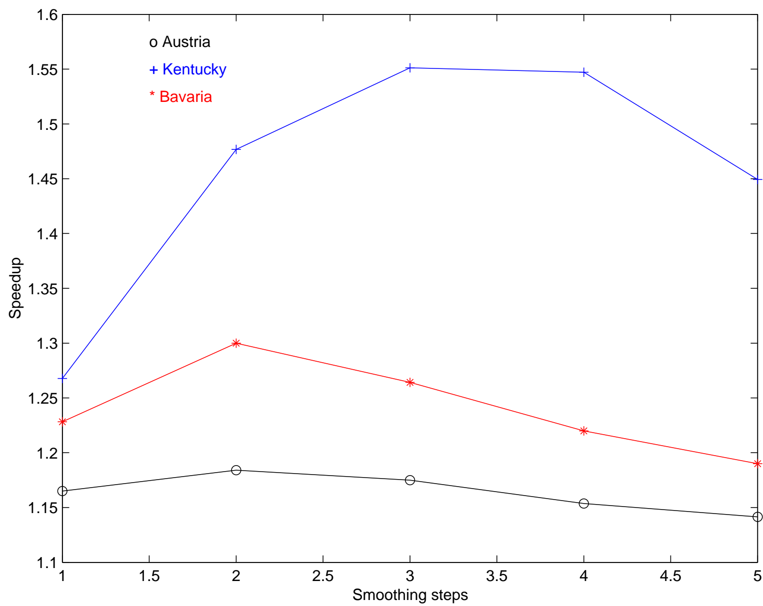
[8] G. Karypis. METIS serial graph partitioning and matrix ordering software. In URL http://www-users.cs.umn.edu/∼karypis/metis/metis/main.shtml, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA.

[9] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *Proceedings of the Seventh ACM Conference on Architectural Support for Programming*

Languages and Operating Systems, pages 60–73, Cambridge, MA, 1996. ACM.



(a) Explicit cache blocking speedups



(b) Active set speedups

FIGURE 4.4. *Speedups on Linux based Pentium III (IA-32)*