

PREPROCESSING COSTS OF CACHE BASED MULTIGRID

CRAIG C. DOUGLAS

*University of Kentucky, Department of Computer Science, 325 McVey Hall - CCS,
Lexington, KY 40506-0045, USA. douglas@ccs.uky.edu*

JONATHAN HU

*University of Kentucky, Mathematics Department, 715 Patterson Office Tower,
Lexington, KY 40506-0027, USA. jhu@ms.uky.edu*

MOHAMED ISKANDARANI

*Institute of Marine and Coastal Sciences, Rutgers University, P.O. Box 231, New
Brunswick, NJ 08903-0231, USA. mohamed@ahab.rutgers.edu*

Multigrid algorithms based on cache aware smoothers produce a high performance, portable, cache aware multigrid solver for problems with one or more degrees of freedom on unstructured grids. This paper analyzes the work cost of a necessary preprocessing step in cache aware multigrid. This mesh renumbering step is not present in standard multigrid codes. The cost is found to be small relative to standard Gauss-Seidel sweeps on the finest multigrid level. The relative cost decreases as the degrees of freedom increase. Numerical experiments demonstrate that the preprocessing cost is less than one standard Gauss-Seidel sweep.

1 Introduction

In Douglas *et al*¹, a Gauss-Seidel algorithm is described which is optimized for memory caches, yet returns bitwise the same solution as a standard Gauss-Seidel algorithm. The cache aware smoother is embedded in a multigrid algorithm to produce a high performance, portable, cache aware multigrid solver for problems with one or more degrees of freedom on unstructured grids.

A critical component of the solver 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. In this paper, we analyze the worst case behavior of the preprocessing algorithms and measure algorithm performance.

Assume that the physical domain $\Omega \subset \mathbb{R}^2$ is simply connected. Assume that 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 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 .

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 $\mathcal{N} := \mathcal{N}(\mathcal{G})$ 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 $(i_1, i_2), (i_2, i_3), \dots, (i_{k-1}, i_k)$, where $i_1 = i$ and $i_k = j$. The *length* of P is $|P| = k + 1$, i.e., one more than the number of edges in P . 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.

A node on the physical boundary of Ω is called a *physical boundary node*. Any node which is not a physical boundary node is a *physical interior node*. Define the set of neighbors of a node u to be $\text{nbr}(u) = \{v \in \mathcal{T} : \text{the discrete solution at } u \text{ depends on the solution at } v\}$. Define the *cache boundary* $\partial\mathcal{G}'$ of subgraph \mathcal{G}' be the set of nodes $\{i \in \mathcal{N}(\mathcal{G}') : \exists j \in \text{nbr}(i) \text{ such that } j \notin \mathcal{N}(\mathcal{G}')\}$. A node which is in a cache boundary set is called a *cache boundary node*. For a given subgraph, any member node which is not a cache boundary node is a *cache interior node*.

The *distance* D_v of a member node v from the cache block boundary is $\min\{|P| : P \text{ is a path from } v \text{ to a boundary node}\}$. Note that 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 $\{v \in \mathcal{N}(\mathcal{S}_i) : D_v = i\}$.

2 Algorithms for Renumbering Nodes

Once the mesh is decomposed into cache blocks, the next step is to determine how many updates are possible on each unknown without referencing data from another block. Within a cache block, the k th *subblock* consists of those nodes which can be updated at most k times without referencing other subblocks. Identifying the number of updates possible for each node in a block without referencing another block is equivalent to identifying the distance D_i of each node i to the cache block boundary. How these distances are found depends upon whether the cache block boundaries are connected and whether physical boundary nodes are known in advance.

The motivation for all of the following algorithms is Dijkstra's method for finding the shortest path between two nodes². A node is *marked* if its distance is known, and a node is *scanned* if all its neighbors' distances are known. In the following analysis, we denote the number of nodes in a cache

block Ω by N_Ω . The total number of nodes in the mesh is N . The degrees of freedom per node is d . We assume that additions, multiplies, divisions, and assignments have equal cost.

2.1 Cache boundaries are connected

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

Alg. 1 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 which 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. 2 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. 2 continues until both stacks are empty. At the conclusion of Alg. 2, vector D contains the smaller of $m + 1$ and the minimum distance from each node of block Ω_s to the boundary $\partial\Omega_s$.

Algorithm 1 (cache boundaries connected) Mark cache boundary nodes.

ALGORITHM LABEL-BOUNDARY-NODES

```

1: Initialize stacks  $S_1$  and  $S_2$ .
2: Set  $D_i = 0$  for all  $i$  in cache block  $\Omega_s$ .
3: Find any node  $i$  on cache boundary  $\partial\Omega_s$ .
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  $\partial\Omega_s$  then
8:     Set  $D_i = 1$ .
9:     for each node  $j$  connected to  $i$  do
10:      if  $D_j == 0$  and  $j$  is in  $\Omega_s$  then
11:        Set  $D_j = -1$ .
12:        Push  $j$  onto  $S_1$ .
13:   else
14:     Set  $D_i = 2$ .
15:     Push  $i$  onto  $S_2$ .

```

Algorithm 2 Mark cache interior nodes.

Label-Internal-Nodes

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

Theorem 2.1 *The cost of Algs. 1 and 2 on the finest grid is at most $\frac{6}{d^2}$ sweeps of standard Gauss-Seidel on the finest grid, where d is the degrees of freedom per node.*

We note that the estimate in Theorem 2.1 is pessimistic. In the proof of 2.1, certain values which are much smaller than N_{Ω_s} are bounded by N_{Ω_s} for convenience.

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

Alg. 3 finds distances of all cache boundary nodes. Alg. 2 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. 1. (See §2.1.) 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 which are distance two from the boundary have been marked and placed on S_2 in preparation for Alg. 2.

Theorem 2.2 *The cost of Algs. 3 and 2 is at most $\frac{7}{d^2}$ sweeps of standard Gauss-Seidel.*

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  $\Omega_s$ .
3: for each node  $i$  in  $\Omega_s$  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  $\Omega_s$ ) then
12:            Set  $D_j = -1$ .
13:            Push  $j$  onto  $S_1$ .
14:          else
15:            Set  $D_i = 2$ .
16:            Push  $i$  onto  $S_2$ .
```

This estimate is pessimistic. We also note that while it appears that Alg. 1 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.

2.3 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 which 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 which 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 which 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  $\Omega_s$ .
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:       while  $S_1 \neq \emptyset$  do
13:         Pop  $i$  off of  $S_1$ .
14:         if  $i$  is in  $\partial\Omega_s$  then
15:           Set  $D_i = 1$ .
16:           for each node  $j$  connected to  $i$  do
17:             if ( $D_j == 0$ ) AND ( $j$  is in  $\Omega_s$ ) then
18:               Set  $D_j = -1$ .
19:               Push  $j$  onto  $S_1$ .
20:           else
21:             Set  $D_i = 2$ .
22:             Push  $i$  onto  $S_2$ .
23:             if  $i$  is a PBN then
24:               Push  $i$  onto  $S_3$ .
25:           until  $S_4$  is empty.
26:       Call Find-Next-Cache-Boundary-Component( $S_3, S_4$ ).
27:   until  $S_4$  is empty

```

Theorem 2.3 *The cost of Algs. 4, 5, and 2 is at most $\frac{3}{d^2} + 1$ sweeps of standard Gauss-Seidel.*

In Theorem 2.3 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.

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  $S_3 \neq \emptyset$  do
2:   Pop  $i$  off of  $S_3$ .
3:   if  $i$  is in  $\partial\Omega_s$  then
4:     Push  $i$  onto  $S_4$ .
5:   for each node  $j$  connected to  $i$  do
6:     if ( $j \in \Omega_s$ ) AND ( $D_j == 0$ ) AND ( $j$  is a PBN) then
7:       Set  $D_j = -1$ .
8:       Push  $j$  onto  $S_3$ .
```

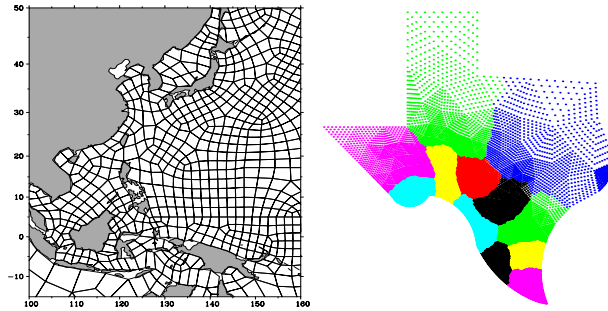


Figure 1. Sample grids.

3 Numerical Results and Conclusions

Experiments were run on on a SGI O2 with a MIPS R12000 300 megahertz IP32 processor, 128 MB of main memory, a 1 MB unified L2 cache, and a 32 KB L1 data cache. Similar results have been found on other RISC computers.

Table 1 compares the CPU time for Algs. 3 and 2 to renumber one mesh of increasing size versus the CPU time required for a standard Gauss-Seidel sweep over the same mesh with and without a residual calculation. Although the problem is a two dimensional Poisson problem, the preprocessing algorithms treat it as a variable coefficient problem on an unstructured grid. It appears that the estimate in Proposition 2.2 is pessimistic. For one degree of freedom, the preprocessing cost is estimated to be no more than seven standard Gauss-Seidel sweeps. In fact, the actual cost in each example is less than the cost of one smoothing step using a traditional implementation. For problems with more than one degree of freedom, the relative cost of preprocessing

Table 1. CPU time (seconds) for Algs. 3 and 2 compared with one sweep of Gauss-Seidel on 2D Poisson problem with one degree of freedom per node.

matrix order	standard GS + residual	standard GS	mesh renumbering
181476	0.54	0.21	0.13
247009	0.59	0.29	0.17
322624	0.87	0.37	0.32
408321	1.09	0.57	0.39

Table 2. CPU times (seconds) for Alg. 3 to renumber 5 meshes with 83178 nodes total with two degrees of freedom per node.

standard GS plus residual	standard GS	mesh renumbering
0.71	0.30	0.09

should be even less than in the first example.

Table 2 compares CPU times on a two dimensional unstructured grid problem with two degrees of freedom per node and approximately 120000 unknowns on the finest level. The time for Algs. 3 and 2 to renumber all five meshes (83178 nodes) is about 30% of the time for one Gauss-Seidel sweep across the finest grid (without a residual calculation).

The preprocessing step outlined in this paper is the only difference between a cache aware multigrid code and a standard multigrid code (except for the smoother itself). The penalty (CPU time) incurred appears to be small compared to the cost of traditional smoothing.

References

1. C. C. Douglas, J. Hu, M. Kowarschik, and U. Rde. Cache optimization for structured and unstructured grid multigrid. *Electron. Trans. Numer. Anal.*, 9, 2000.
2. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.