UC San Diego
Technical Reports

Title
Proof of Correctness for Sparse Tiling of Gauss-Seidel

Permalink
https://escholarship.org/uc/item/14z6d0p3

Authors
Strout, Michelle Mills
Carter, Larry
Ferrante, Jeanne

Publication Date
2001-12-04

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Proof of Correctness for Sparse Tiling of Gauss-Seidel

Michelle Mills Strout, Larry Carter, and Jeanne Ferrante
University of California, San Diego

UCSD Department of Computer Science and Engineering
Technical Report #CS2001-0686
November 2001

Abstract

Gauss-Seidel is an iterative computation used for solving sets of simulataneous linear equations, $Au = f$. When these unknowns are associated with nodes in an irregular mesh, then the Gauss-Seidel computation structure is related to the mesh structure. We use this structure to subdivide the computation at runtime using a technique called *sparse tiling*. The rescheduled computation exhibits better data locality and therefore improved performance. This paper gives a complete proof that a serial schedule based on sparse tiling generates results equivalent to those that a standard Gauss-Seidel computation produces.

1 Introduction

Gauss-Seidel and Jacobi are examples of iterative methods that are used to solve simultaneous linear equations, $Au = f$. Iterative methods solve for u by iterating over the system of equations, converging towards a solution. The iteratively calculated value of a mesh node unknown u_j depends on the values of other unknowns on the same node, the unknowns associated with adjacent nodes within the mesh, and the non-zero coefficients in the sparse matrix which relate those unknowns. Typically the sparse matrix is so large that none of the values used by one calculation of u_j remain in the cache for future iterations on u_j; thus the computation exhibits poor data locality.

The pseudo-code for Gauss-Seidel is shown in (1). For each iteration of the outermost loop, the entire sparse matrix is traversed. We refer to the iterator i of this outermost loop as the *convergence iterator*. The j loop iterates over the rows in the sparse matrix.[1] The k loop, which is implicit in the summations, iterates over the unknowns related to u_j, with $a_{jk}u_k^{(i)}$ and $a_{jk}u_k^{(i-1)}$ only being computed when a_{jk} is a non-zero matrix value. a_{jk} is the entry in the sparse matrix A at row j and column k. At each convergence iteration a new value is generated for each unknown u_j, using the most recently calculated values for neighboring unknowns.

```
for $i = 1, 2, ..., T$
    for $j = 1, 2, ..., R$
        $u_j^{(i)} = (1/a_{jj})(f_j - \sum_{k=1}^{j-1} a_{jk}u_k^{(i)} - \sum_{k=j+1}^{R} a_{jk}u_k^{(i-1)})$        (1)
```

[1]There is one row in the matrix for each unknown at each mesh node.

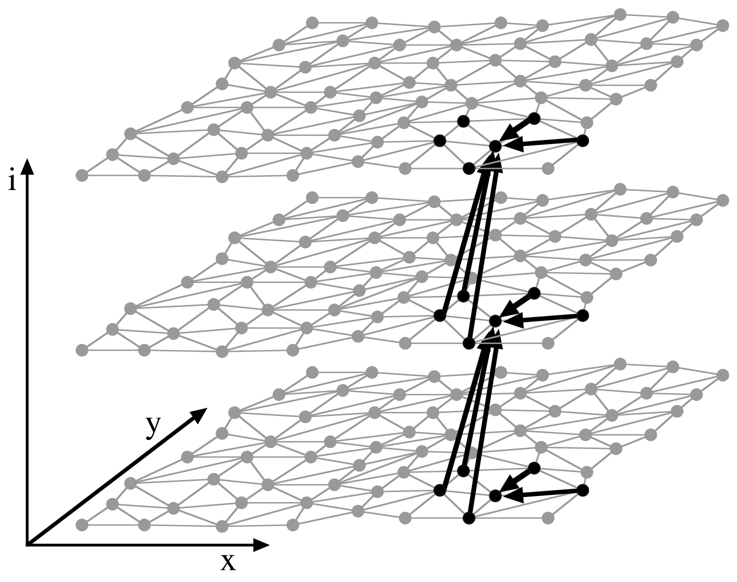

Figure 1: Irregular Gauss-Seidel iteration space graph for 3 convergence iterations

The Gauss-Seidel computation over an irregular 2D mesh can be visualized with the iteration space graph shown in figure 1. The iteration space shown contains three convergence iterations. Each black iteration point[2] , $<i, v>$, represents the computations for all $u_j^{(i)}$ where u_j is an unknown associated with mesh node v and i is the convergence iteration. The arrows represent data dependences[3] between the iteration points. Gauss-Seidel uses the most recent version of its neighbors, so some data dependences come from points in the same i iteration and some data dependences come from points in the previous i iteration.

In this setting, data locality means that the data needed for the current iteration point are already in cache. If data used or generated by neighboring iteration points in the same i iteration are in cache, then the computation exhibits *intra-iteration* locality. *Inter-iteration* locality occurs when the data from previous i iterations remain in cache until their final use. Typical implementations of iterative algorithms like Gauss-Seidel follow the schedule seen in (1), where the entire sparse matrix associated with the mesh is traversed for each convergence iteration. Because the mesh and therefore associated sparse matrix are typically quite large it is very unlikely that inter-iteration locality occurs in a typical iterative solver implementation. Furthermore, the order in which iteration points in the same i iteration are visited will affect the amount of intra-iteration locality.

Tiling is a compile-time transformation which subdivides the iteration space for a regular computation so that the new tile-based schedule, where each tile is executed atomically, exhibits better data locality. We say a tile is executed atomically when all the iteration points in the tile are executed before any iteration points in subsequent tiles. The non-affine loop bounds and indirect memory references in sparse matrix computations prohibit the use of compile time transformations. We have developed *sparse tiling* [18], which tiles the iteration space resulting from an irregular mesh at run-time (see figure 2). The sparse tiled iteration space is then used to guide run-time rescheduling and data reordering of Gauss-Seidel. Specifically, the schedule changes from sweeping over the entire mesh each convergence iteration to executing the iteration points tile-by-tile.

Sparse tiling uses a graph partitioner to divide the irregular mesh into approximately equal size cells. In

[2]We use the term iteration point for points in the iteration space graph and node for points in the mesh.
[3]Only the dependences for one mesh node are shown for clarity.

figure 2 the partitioning logically occurs at the middle convergence iteration, $i = 2$. The cells in the partition act as seeds for tiles which are grown throughout the rest of the iteration space. For Gauss-Seidel over irregular 2D meshes, this results in irregular column-shaped tiles which include iteration points from all convergence iterations. The new schedule using sparse tiles improves inter- and intra-iteration locality and thereby improve performance [18].

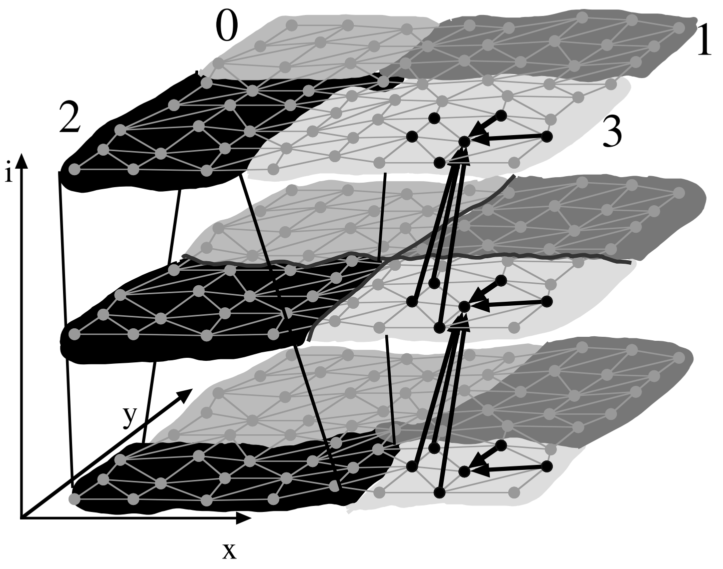

Figure 2: Sparse tiled Gauss-Seidel

In this paper, we describe the structure of a Gauss-Seidel computation over an unstructure mesh. We then give a run-time sparse tiling algorithm for such a computation. Based on the resulting sparse tiling we are able to construct a new execution which satisfies the Gauss-Seidel partial ordering constraints.

2 Terminology

The mesh can be represented by a graph $G(V, E)$ consisting of a set of nodes V and edges E. Although a mesh typically has undirected edges $(v, w) \in E$, for presentation purposes assume that for each undirected edge (v, w) in the mesh there are two directed edges $<v, w> \in E$ and $<w, v> \in E$. Recall that each node in a mesh can have multiple unknowns associated with it such as temperature, displacement, pressure, etc. The sparse matrix A is generated by creating a linear equation for each unknown on each node in the mesh. Assuming each unknown has a non-zero coefficient in the sparse matrix A for all other unknowns on the same node including itself and all unknowns on neighboring nodes, then the size of the sparse matrix is $d^2(|E| + |V|)$ where d is the number of unknowns or degrees of freedom for each mesh node.

To describe the sparse tiling algorithm we use the following terminology.

Iteration point, $<i, v>$, represents all computation at convergence iteration i for the unknowns associated with node v.

Partition function, $part(v) : V \to \{0, ..., (k-1)\}$, maps each node to a cell of the mesh partitioning, where k is the number of tiles.

Tiling function, $\theta(i, v) : \{1, .., T\}$ x $V \to \{0, ..., (k-1)\}$, returns the tile identifier of the tile responsible for executing the given iteration point. The tile identifier also indicates the execution order of the tiles.

Reordering function, $\sigma(v) : V \to \{0, ..., (|V| - 1)\}$, specifies a complete ordering of the nodes $v \in V$.

Schedule, $s(t, i) : \{0, ..., (k-1)\}$ x $\{1, ..., T\} \to 2^{\{0, ..., (R-1)\}}$, specifies for each tile and convergence iteration the subset of matrix rows which need to be computed, where T is the number of convergence iterations.

Execution, $e(i, v) : \{1, ..., T\}$ x $V \to \{1, 2, ..., (T * |V|)\}$, indicates the relative execution time for each iteration point.

3 Gauss-Seidel Partial Ordering Constraints

The outline of a Gauss-Seidel computation is as follows:

1. Choose an arbitrary order for the nodes in the mesh, $\sigma(v) : V \to \{0, ..., (|V| - 1)\}$.
2. Iterate towards convergence over the mesh T times, where each iteration visits all the nodes in the chosen order

During each convergence iteration the unknowns at each node v are updated using the most recent values of unknowns associated with neighboring nodes w where $<v, w> \in E$. The execution resulting from a typical Gauss-Seidel schedule is $e(i, v) = (i-1)|V| + \sigma(v)$.

By analyzing the structure of a Gauss-Seidel computation, we find that although the Gauss-Seidel schedule gives a total ordering on the execution, the data dependences between various parts of the computation allow for a less strict partial order. If the partial order is satisfied by another execution, then given the same node order $\sigma(v)$, the new execution will generate a solution which is bitwise identical to the original Gauss-Seidel execution.

A Gauss-Seidel execution satisfies the following **Partial Ordering Constraints**.

1. $\forall i : 1 \leq i \leq (T-1)$ and $\forall v \in V$, $e(i, v) < e(i+1, v)$
   (the iteration points for each individual mesh node are executed in order)
2. $\forall i : 1 \leq i \leq (T-1)$ and $\forall <v, w> \in E$, if $\sigma(v) < \sigma(w)$ then $e(i, v) < e(i, w) < e(i+1, v)$
   (if there is an edge between v and w then the execution must maintain the order between v and w specified by σ at each iteration, and later iterations of the node v must execute after previous iterations of w)

4 Tiling Gauss-Seidel

Sparse tiling [18] subdivides the iteration space of iterative algorithms such as Gauss-Seidel in order to do run-time data reordering and rescheduling. Sparse tiling includes the following run-time steps.

- **Partition** the mesh
- **Tile** the iteration space
- **Reorder** mesh nodes to improve intra-iteration locality
- **Generate** the sparse matrix from the reordered mesh
- **Reschedule** the sparse matrix computation to improve inter-iteration locality
- **Execute** the new schedule on the sparse matrix

The next sub-sections describe each part of the run-time process for *sparse tiling.* Finally, we prove that a serial execution based on sparse tiling satisfies the Gauss-Seidel partial ordering constraints.

4.1 Partition the Mesh

Although optimal graph partitioning is an NP-Hard problem [5], there are many heuristics used to get reasonable graph partitions. The goal of graph partitioning is to divide the nodes of a graph into k roughly equal-sized cells, in a way that minimizes the number of edges whose two endpoints are in different cells. Currently, we use the kmetis paritioning function in Metis which is based on the multilevel k-way partitioning described in [11]. The k-way partitioning algorithm has a complexity of $O(|E|)$, where $|E|$ is the number of edges in the mesh. After the partitioning, all mesh nodes $v \in V$ have been assigned a cell in the partitioning, $part(v)$.

4.2 Tile the Iteration Space

The mesh partitioning, generated in the **Partition** step, creates a seed partitioning from which tiles can be grown. We use the seed partitioning as the tiling at a particular convergence iteration, i_s. In other words at i_s, where i_s is one of the convergence iterations 1 through T, $\theta(i_s, v) = part(v)$. To determine the tiling at other convergence iterations we add or delete iteration points from the seed partition to allow atomic execution of tiles across convergence iterations without violating any data dependences.

The SparseNaive Algorithm shown in figure 3 will generate the tiling function θ, which assigns iteration points to tiles, and the relation $NodeOrd$, which specifies some constraints on the reordering function σ. The first three instructions initialize the $NodeOrd$ relation and all of the θ's for the convergence iteration i_s. We then loop down through the convergence iterations that come before i_s setting the θ function for each iteration point $<i, v>$ to the same as the iteration point directly above it. Finally, we visit the edges in the mesh adjusting θ to ensure that the data dependences are satisfied by the θ values. We repeat this process for the convergence iterations between i_s and T in the upwards tiling. Once neighboring nodes are put into two different tiles at any iteration i, there must be a constraint on their node order σ which we indicate by putting $<v, w>$ into the relation $NodeOrd$ if for any iteration i, $\theta(i, v) < \theta(i, w)$.

An upper bound on the complexity of this algorithm is $O(Tk|V||E|)$, where T is the number of convergence iterations, k is the number of tiles, $|V|$ is the number of nodes in the mesh, and $|E|$ is the number of edges in the mesh. The $k|V||E|$ term is due to the while loops at lines 6 and 17. In the worst-case, the while loop will execute $k|V|$ times. $\forall v \in V$, $\theta(i, v)$ decreases monotonically and can take on at most k values, and in the worse case only one $\theta(i, v)$ would decrease each time through the while loop. In practice, the algorithm runs much faster than this bound.

```
Algorithm SparseNaive($G(V,E)$,$part$,$T$,$i_s$,$m$)

 1:  $\forall v \in V, \theta(i_s, v) \leftarrow part(v)$
 2:  for $1 \leq i \leq T$, $NodeOrd(i) \leftarrow \emptyset$
 3:  $NodeOrd(i_s) \leftarrow \{<v,w> \mid \theta(i_s,v) < \theta(i_s,w)$ and $<v,w> \in E\}$

Downwards tile growth
 4:  for $i = (i_s - 1)$ downto 1
 5:      foreach vertex $v \in V$, $\theta(i,v) \leftarrow \theta(i+1,v)$ end foreach
 6:      do while $\theta$ changes
 7:          foreach $<v,w> \in NodeOrd(i+1)$
 8:              $\theta(i,w) \leftarrow min(\theta(i,w), \theta(i+1,v))$
 9:              $\theta(i,v) \leftarrow min(\theta(i,v), \theta(i,w))$
 10:         end foreach
 11:     end do while
 12:     $NodeOrd(i) \leftarrow NodeOrd(i+1) \bigcup \{<v,w> \mid \theta(i,v) < \theta(i,w)$ and $<v,w> \in E\}$
 13: end for

 14: $NodeOrd(i_s) \leftarrow NodeOrd(1)$

Upwards tile growth
 15: for $i = (i_s + 1)$ to $T$
 16:     foreach vertex $v \in V$, $\theta(i,v) \leftarrow \theta(i-1,v)$ end foreach
 17:     do while $\theta$ changes
 18:         foreach $<v,w> \in NodeOrd(i-1)$
 19:             $\theta(i,v) \leftarrow max(\theta(i,v), \theta(i-1,w))$
 20:             $\theta(i,w) \leftarrow max(\theta(i,w), \theta(i,v))$
 21:         end foreach
 22:     end do while
 23:     $NodeOrd(i) \leftarrow NodeOrd(i-1) \bigcup \{<v,w> \mid \theta(i,v) < \theta(i,w)$ and $<v,w> \in E\}$
 24: end for

 25: $NodeOrd \leftarrow NodeOrd(T)$
```

Figure 3: SparseNaive Pseudocode

```
Algorithm SparseNaive($G(V,E)$,$part$,$T$,$i_s$,$m$)

      { pre-condition [1.1] $(1 \leq T)$ and $(1 \leq i_s \leq T)$ and $(2 \leq m \leq |V|)$}

 1:   $\forall v \in V, \theta(i_s, v) \leftarrow part(v)$
 2:   for $1 \leq i \leq T$, $NodeOrd(i) \leftarrow \emptyset$
 3:   $NodeOrd(i_s) \leftarrow \{<v,w> \mid \theta(i_s,v) < \theta(i_s,w)$ and $<v,w> \in E\}$

      { post-condition [3.1] $\forall v \in V, \theta(i_s, v)$ is initialized }
      { post-condition [3.2] $\theta(i_s,v) < \theta(i_s,w)$ if and only if $<v,w> \in NodeOrd(i_s)$}

Downwards tile growth
 4:   for $i = (i_s - 1)$ downto 1

            { pre-condition [5.1] $\forall v \in V, \theta(i+1, v)$ is initialized }
 5:         foreach vertex $v \in V$, $\theta(i,v) \leftarrow \theta(i+1,v)$ end foreach
            { post-condition [5.1] $\forall v \in V$, $\theta(i,v) = \theta(i+1,v)$}

 6:         do while $\theta$ changes
 7:               foreach $<v,w> \in NodeOrd(i+1)$
 8:                     $\theta(i,w) \leftarrow min(\theta(i,w), \theta(i+1,v))$
                        { post-condition [8.1] $\theta(i,w) \leq \theta(i+1,w)$}
                        { post-condition [8.2] $\theta(i,w) \leq \theta(i+1,v)$}

 9:                     $\theta(i,v) \leftarrow min(\theta(i,v), \theta(i,w))$
                        { post-condition [9.1] $\theta(i,v) \leq \theta(i+1,v)$}
                        { post-condition [9.2] $\theta(i,v) \leq \theta(i,w)$}

10:               end foreach
                  { post-condition [10.1] $\forall v \in V$, $\theta(i,v) \leq \theta(i+1,v)$}
                  { post-condition [10.2] if $\theta$ didn't change then
                        $\forall <v,w> \in NodeOrd(i+1)$, $\theta(i,v) \leq \theta(i,w) \leq \theta(i+1,v)$}

11:         end do while
            { post-condition [11.1] $\forall <v,w> \in NodeOrd(i+1)$, $\theta(i,v) \leq \theta(i,w) \leq \theta(i+1,v)$}
            { post-condition [11.2] $\forall <v,w> \in E$, $\theta(i,v) \leq \theta(i+1,w)$}

12:         $NodeOrd(i) \leftarrow NodeOrd(i+1) \bigcup \{<v,w> \mid \theta(i,v) < \theta(i,w)$ and $<v,w> \in E\}$
            { post-condition [12.1] $\forall <v,w> \in E$, if $\theta(i,v) < \theta(i,w)$ then $<v,w> \in NodeOrd(i)$}
            { post-condition [12.2] $NodeOrd(i+1) \subseteq NodeOrd(i)$}

13: end for
      { post-condition [13.1] $\forall i : 1 \leq i \leq (i_s - 1)$ and $\forall v \in V$, $\theta(i,v) \leq \theta(i+1,v)$}
      { post-condition [13.2] $\forall i : 1 \leq i \leq (i_s - 1)$ and $\forall <v,w> \in E$, $\theta(i,v) \leq \theta(i+1,w)$}
      { post-condition [13.3] $\forall i : 1 \leq i \leq (i_s - 1)$, $NodeOrd(i)$ is acyclic }
      { post-condition [13.4] $\forall i : 1 \leq i \leq (i_s - 1)$ and $\forall <v,w> \in E$,
            if $\theta(i,v) < \theta(i,w)$ then $<v,w> \in NodeOrd(i)$ }
```

Figure 4: SparseNaive Pseudocode with post-conditions, Part I

```
Algorithm SparseNaive cont...

14: $NodeOrd(i_s) \leftarrow NodeOrd(1)$
      { post-condition [14.1] for $i = i_s$, $NodeOrd(i)$ is acyclic }
      { post-condition [14.2] for $i = i_s$ and $\forall <v,w> \in E$,
            if $\theta(i,v) < \theta(i,w)$ then $<v,w> \in NodeOrd(i)$}

Upwards tile growth
15: for $i = (i_s + 1)$ to $T$

            { pre-condition [16.1] $\forall v \in V$, $\theta(i-1,v)$ is initialized }
16:         foreach vertex $v \in V$, $\theta(i,v) \leftarrow \theta(i-1,v)$ end foreach
            { post-condition [16.1] $\forall v \in V$, $\theta(i,v) = \theta(i-1,v)$}

17:         do while $\theta$ changes
18:               foreach $<v,w> \in NodeOrd(i-1)$
19:                     $\theta(i,v) \leftarrow max(\theta(i,v), \theta(i-1,w))$
                        { post-condition [19.1] $\theta(i-1,v) \le \theta(i,v)$}
                        { post-condition [19.2] $\theta(i-1,w) \le \theta(i,v)$}

20:                     $\theta(i,w) \leftarrow max(\theta(i,w), \theta(i,v))$
                        { post-condition [20.1] $\theta(i-1,w) \le \theta(i,w)$}
                        { post-condition [20.2] $\theta(i,v) \le \theta(i,w)$}

21:               end foreach
                  { post-condition [21.1] $\forall v \in V$, $\theta(i-1,v) \le \theta(i,v)$}
                  { post-condition [21.2] if $\theta$ didn't change then
                        $\forall <v,w> \in NodeOrd(i-1)$, $\theta(i-1,w) \le \theta(i,v) \le \theta(i,w)$}

22:         end do while
            { post-condition [22.1] $\forall <v,w> \in NodeOrd(i-1)$, $\theta(i-1,w) \le \theta(i,v) \le \theta(i,w)$}
            { post-condition [22.2] $\forall <v,w> \in E$, $\theta(i-1,v) \le \theta(i,w)$}

23:         $NodeOrd(i) \leftarrow NodeOrd(i-1) \bigcup \{<v,w> \mid \theta(i,v) < \theta(i,w) \text{ and } <v,w> \in E\}$
            { post-condition [23.1] $\forall <v,w> \in E$, if $\theta(i,v) < \theta(i,w)$ then $<v,w> \in NodeOrd(i)$}

24: end for
      { post-condition [24.1] $\forall q : (i_s+1) \le q \le T$ and $\forall v \in V$, $\theta(q-1,v) \le \theta(q,v)$}
      { post-condition [24.2] $\forall q : (i_s+1) \le q \le T$ and $\forall <v,w> \in E$, $\theta(q-1,v) \le \theta(q,w)$}
      { post-condition [24.3] $\forall i : (i_s+1) \le i \le T$, $NodeOrd(i)$ is acyclic }
      { post-condition [24.4] $\forall i : (i_s+1) \le i \le T$ and $\forall <v,w> \in E$,
            if $\theta(i,v) < \theta(i,w)$ then $<v,w> inNodeOrd(i)$}

25: $NodeOrd \leftarrow NodeOrd(T)$

      { post-condition [25.1] $\forall i : 1 \le i \le (T-1)$ and $\forall v \in V$, $\theta(i,v) \le \theta(i+1,v)$}
      { post-condition [25.2] $\forall i : 1 \le i \le (T-1)$ and $\forall <v,w> \in E$, $\theta(i,v) \le \theta(i+1,w)$}
      { post-condition [25.3] $\forall i : 1 \le i \le T$, $NodeOrd(i)$ is acyclic }
      { post-condition [25.4] $\forall i : 1 \le i \le T$ and $\forall <v,w> \in E$,
            if $\theta(i,v) < \theta(i,w)$ then $<v,w> \in NodeOrd(i)$}
```

Figure 5: SparseNaive Pseudocode with post-conditions, Part II

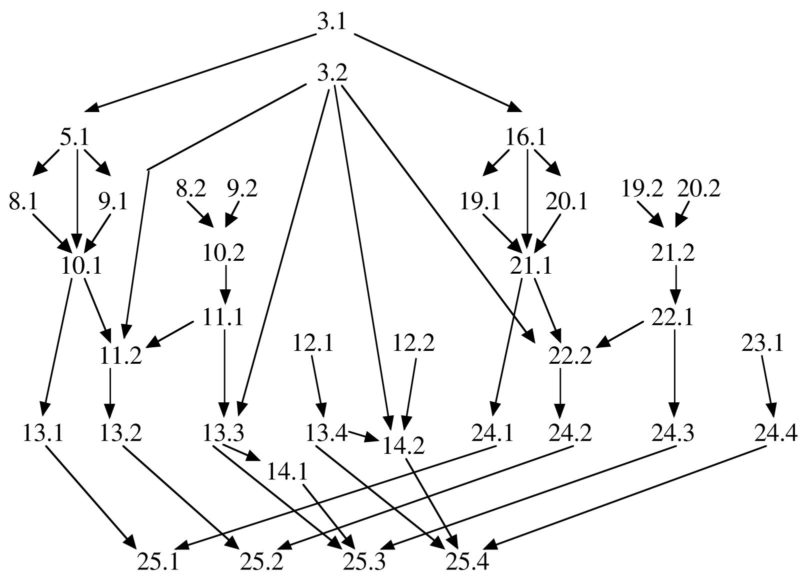

Figure 6: Dependences between post-conditions for SparseNaive Algorithm

Figures 4 and 5 list post-conditions for the SparseNaive tiling algorithm. Figure 6 shows how the post-conditions relate to one another. Next we will give a proof for each of the post-conditions.

SparseNaive Post-condition 3.1 $\forall v \in V, \theta(i_s, v)$ *is initialized*

Satisfied by assignments in line 1.

SparseNaive Post-condition 3.2 $\theta(i_s, v) < \theta(i_s, w)$ *if and only if* $<v, w> \in NodeOrd(i_s)$

Satisfied by assignments in lines 2 and 3.

SparseNaive Post-condition 5.1 $\forall v \in V,\ \theta(i, v) = \theta(i+1, v)$

Satisfied by assignment in line 5 and pre-condition 5.1. Pre-condition 5.1 is satisfied by post-condition 3.1 when $i = (i_s - 1)$ in the loop starting at line 4. For all i such that $1 \leq i < (i_s - 1)$, it is satisfied by the previous loop iteration's post-condition 5.1.

SparseNaive Post-condition 8.1 $\theta(i, w) \leq \theta(i+1, w)$

Post-condition 5.1 ensures that $\theta(i, w)$ starts out equal to $\theta(i+1, w)$. In line 8, $\theta(i, w)$ can only be reduced thus postcondition 8.1 holds.

SparseNaive Post-condition 8.2 $\theta(i, w) \leq \theta(i+1, v)$

Satisfied by the assignment in line 8.

SparseNaive Post-condition 9.1 $\theta(i,v) \leq \theta(i+1,v)$

Post-condition 5.1 ensures that $\theta(i,v)$ starts out equal to $\theta(i+1,v)$. In line 9, $\theta(i,v)$ can only be reduced thus postcondition 9.1 holds.

SparseNaive Post-condition 9.2 $\theta(i,v) \leq \theta(i,w)$

Satisfied by the assignment in line 9.

SparseNaive Post-condition 10.1 *$\forall v \in V$, $\theta(i,v) \leq \theta(i+1,v)$*

Satisfied by post-conditions 5.1, 8.1, and 9.1.

SparseNaive Post-condition 10.2 *if θ didn't change then $\forall <v,w>\in NodeOrd(i+1)$, $\theta(i,v) \leq \theta(i,w) \leq \theta(i+1,v)$*

Follows immediately from post-conditions 8.2 and 9.2. Notice that it is important that θ not change during the entire foreach loop for this post-condition to be true. For example, assume that $<v_1,v_2>\in NodeOrd(i+1)$ and $<v_2,v_3>\in NodeOrd(i+1)$ with $\theta(i+1,v_3)=0, \theta(i,v_1)=\theta(i+1,v_1)>0$, and $\theta(i,v_2)>0$. If edge $<v_1,v_2>$ is visited first in the foreach loop, then it will still be the case that $\theta(i,v_1)>0$ and $\theta(i,v_2)>0$. However, later in the foreach loop when $<v_2,v_3>$ is visited, $\theta(i,v_2)$ will be set equal to 0 due to line 9. It will then be the case that $\theta(i,v_1)>\theta(i,v_2)$ which will be remedied the next time through the foreach loop.

SparseNaive Post-condition 11.1 *$\forall <v,w>\in NodeOrd(i+1)$, $\theta(i,v) \leq \theta(i,w) \leq \theta(i+1,v)$*

The do while loop in lines 6 through 11 ends when θ no longer changes in the foreach loop starting at line 7. Therefore, due to post-condition 10.2, post-condition 11.1 is satisfied.

SparseNaive Post-condition 11.2 *$\forall <v,w>\in E$, $\theta(i,v) \leq \theta(i+1,w)$*

For $i=(i_s-1)$ and $\forall <v,w>\in E$, the tiling function values $\theta(i+1,v)$ and $\theta(i+1,w)$ were set in line 1. For all i such that $1 \leq i < (i_s-1)$ and $\forall <v,w>\in E$, the tiling function values $\theta(i+1,v)$ and $\theta(i+1,w)$ were set in the previous iteration of the for loop starting at line 4. The relationship between $\theta(i+1,v)$ and $\theta(i+1,w)$ falls under three cases, either $\theta(i+1,v) < \theta(i+1,w)$, $\theta(i+1,v) > \theta(i+1,w)$, or $\theta(i+1,v) = \theta(i+1,w)$.

Case 1: If $\theta(i+1,v) < \theta(i+1,w)$ then due to post-condition 3.2 if $i=(i_s-1)$ and post-condition 12.1 if $1 \leq i < (i_s-1)$, the following is true.

$$<v,w>\in NodeOrd(i+1) \tag{2}$$

Due to (2), post-condition 10.1, and the first inequality in post-condition 11.1, the following is true for all i such that $1 \leq i \leq (i_s - 1)$.

$$\theta(i,v) \leq \theta(i,w) \leq \theta(i+1,v) \tag{3}$$

Case 2: If $\theta(i+1,v) > \theta(i+1,w)$ then due to post-condition 3.2 if $i = (i_s - 1)$ and post-condition 12.1 if $1 \leq i < (i_s - 1)$, the following is true.

$$<v,w> \in NodeOrd(i+1) \tag{4}$$

Using (4) and swapping the roles of v and w in post-condition 11.1, we find the following is true for all i such that $1 \leq i \leq (i_s - 1)$.

$$\theta(i,w) \leq \theta(i,v) \leq \theta(i+1,w) \tag{5}$$

Case 3: If $\theta(i+1,v) = \theta(i+1,w)$ then due to post-condition 10.1 the following is true for all i such that $1 \leq i \leq (i_s - 1)$.

$$\theta(i,v) \leq \theta(i+1,v) = \theta(i+1,w) \tag{6}$$

SparseNaive Post-condition 12.1 $\forall <v,w> \in E$, *if* $\theta(i,v) < \theta(i,w)$ *then* $<v,w> \in NodeOrd(i)$

Satisfied by the assignment in line 12.

SparseNaive Post-condition 12.2 $NodeOrd(i+1) \subseteq NodeOrd(i)$

Satisfied by the assignment in line 12.

SparseNaive Post-condition 13.1 $\forall i : 1 \leq i \leq (i_s - 1)$ *and* $\forall v \in V$, $\theta(i,x) \leq \theta(i+1,v)$

Satisfied by the loop bounds of the for loop starting at line 4 and post-condition 10.1.

SparseNaive Post-condition 13.2 $\forall i : 1 \leq i \leq (i_s - 1)$ *and* $\forall <v,w> \in E$, $\theta(i,v) \leq \theta(i+1,w)$

Satisfied by the loop bounds of the for loop starting at line 4 and post-condition 11.2.

SparseNaive Post-condition 13.3 $\forall i : 1 \leq i \leq (i_s - 1)$, $NodeOrd(i)$ *is acyclic*

During the downwards tile growth, relations are added to $NodeOrd(i)$ at line 12. Post-condition 3.2 quarantees that $NodeOrd(i_s)$ is initialized as acyclic. Using the inductive assumption that $NodeOrd(i+1)$ is acyclic, we show that each new relation added at line 12 does not cause a cycle with the relations in $NodeOrd(i+1)$ and any other relations previously added during the current execution of line 12.

We assume the contrary and then derive a contradiction. Assume there is a path $<w,x_0> ... <x_n,v>$ in the current set of relations such that upon adding $<v,w>$ a cycle would be created. Due to the first inequality in post-condition 11.1 and line 12 the following statement is true about the tiling function θ values for the nodes in the path at the current convergence iteration i.

$$\theta(i,w) \leq \theta(i,x_0) \leq \cdots \leq \theta(i,x_n) \leq \theta(i,v) \tag{7}$$

Due to line 12, if the relation $<v,w>$ is being added to $NodeOrd(i)$ then the following is true.

$$\theta(i,v) < \theta(i,w) \tag{8}$$

Combining (7) and (8) results in the contradiction that $\theta(i,v) < \theta(i,v)$. Therefore, it is not possible to add a relation $<v,w>$ to $NodeOrd(i)$ which will cause a cycle.

SparseNaive Post-condition 13.4 *$\forall i : 1 \leq i \leq (i_s - 1)$ and $\forall <v,w> \in E$, if $\theta(i,v) < \theta(i,w)$ then $<v,w> \in NodeOrd(i)$*

Satisfied by the loop bounds of the for loop starting at line 4 and post-condition 12.1.

SparseNaive Post-condition 14.1 *for $i = i_s$, $NodeOrd(i)$ is acyclic*

Satisfied by post-condition 13.3 and the assignment in line 14.

SparseNaive Post-condition 14.2 *for $i = i_s$ and $\forall <v,w> \in E$, if $\theta(i,v) < \theta(i,w)$ then $<v,w> \in NodeOrd(i)$*

Due to post-condition 12.2, the following statement is true.

$$NodeOrd(i_s) \subseteq NodeOrd(i_s - 1) \subseteq \cdots \subseteq NodeOrd(1) \tag{9}$$

Therefore after the assignment in line 14, the relations put into $NodeOrd(i_s)$ at line 3 are still in $NodeOrd(i_s)$ and post-condition 14.2 is satisfied by post-condition 3.2.

SparseNaive Post-condition 16.1 *$\forall v \in V$, $\theta(i,v) = \theta(i-1,v)$*

Satisfied by assignment in line 16 and pre-condition 16.1. Pre-condition 16.1 is satisfied by post-condition 3.1 when $i = (i_s - 1)$ in the loop starting at line 15. For all i such that $1 \leq i < (i_s - 1)$, it is satisfied by the previous loop iteration's post-condition 16.1.

SparseNaive Post-condition 19.1 *$\theta(i-1,v) \leq \theta(i,v)$*

Post-condition 16.1 ensures that $\theta(i,v)$ starts out equal to $\theta(i-1,v)$. In line 19, $\theta(i,v)$ can only be increased in value thus postcondition 19.1 holds.

SparseNaive Post-condition 19.2 *$\theta(i-1,w) \leq \theta(i,v)$*

Satisfied by the assignment in line 19.

SparseNaive Post-condition 20.1 $\theta(i-1,w) \leq \theta(i,w)$

Post-condition 16.1 ensures that $\theta(i,w)$ starts out equal to $\theta(i-1,w)$. In line 20, $\theta(i,w)$ can only be increased in value thus postcondition 20.1 holds.

SparseNaive Post-condition 20.2 $\theta(i,v) \leq \theta(i,w)$

Satisfied by the assignment in line 20.

SparseNaive Post-condition 21.1 *$\forall v \in V$, $\theta(i-1,v) \leq \theta(i,v)$*

Satisfied by post-conditions 16.1, 19.1, and 20.1.

SparseNaive Post-condition 21.2 *if θ didn't change then $\forall <v,w> \in NodeOrd(i-1)$, $\theta(i-1,w) \leq \theta(i,v) \leq \theta(i,v)$*

Follows immediately from post-conditions 19.2 and 20.2. Notice that it is important that θ not change during the entire foreach loop for this post-condition to be true. For example, assume that $<w_1,w_2> \in NodeOrd(i-1)$ and $<w_2,w_3> \in NodeOrd(i-1)$ with $\theta(i-1,w_1) = \theta(i,w_1) = 4, \theta(i-1,w_2) = \theta(i,w_2) < 4$, and $\theta(i,w_3) < 4$. If edge $<w_2,w_3>$ is visited first in the foreach loop, then after lines 19 and 20 it will still be the case that $\theta(i,w_2) < 4$ and $\theta(i,w_3) < 4$. However, later in the foreach loop when $<w_1,w_2>$ is visited, $\theta(i,w_2)$ will be set equal to 4 due to line 20. It will then be the case that $\theta(i,w_2) > \theta(i,w_3)$ which will be remedied the next time through the foreach loop.

SparseNaive Post-condition 22.1 *$\forall <v,w> \in NodeOrd(i-1)$, $\theta(i-1,w) \leq \theta(i,v) \leq \theta(i,w)$*

The do while loop in lines 17 through 22 ends when θ no longer changes in the foreach loop starting at line 18. Therefore, due to post-condition 21.2, post-condition 22.1 is satisfied.

SparseNaive Post-condition 22.2 *$\forall <v,w> \in E$, $\theta(i-1,v) \leq \theta(i,w)$*

For $i = (i_s+1)$ and $\forall <v,w> \in E$, the tiling function values $\theta(i-1,v)$ and $\theta(i-1,w)$ were set in line 1. For all i such that $(i_s+1) < i \leq T$ and $\forall <v,w> \in E$, the tiling function values $\theta(i-1,v)$ and $\theta(i-1,w)$ were set in the previous iteration of the for loop starting at line 15. The relationship between $\theta(i-1,v)$ and $\theta(i-1,w)$ falls under three cases, either $\theta(i-1,v) < \theta(i-1,w)$, $\theta(i-1,v) > \theta(i-1,w)$, or $\theta(i-1,v) = \theta(i-1,w)$.

Case 1: If $\theta(i-1,v) < \theta(i-1,w)$ then due to post-conditions 3.2 and 12.2 if $i = (i_s+1)$ and post-condition 23.1 if $(i_s+1) < i \leq T$, the following is true.

$$<v,w> \in NodeOrd(i-1) \tag{10}$$

Due to (10), post-condition 21.1, and the second inequality in post-condition 22.1, the following is true for all i such that $(i_s+1) \leq i \leq T$.

$$\theta(i-1,v) \leq \theta(i,v) \leq \theta(i,w) \tag{11}$$

Case 2: If $\theta(i-1,v) > \theta(i-1,w)$ then due to post-conditions 3.2 and 12.2 if $i = (i_s+1)$ and post-condition 23.1 if $(i_s+1) < i \leq T$, the following is true.

$$<w,v> \in NodeOrd(i-1) \tag{12}$$

Using (12) and swapping the roles of v and w in post-condition 22.1, we find the following is true for all i such that $(i_s+1) \leq i \leq T$.

$$\theta(i-1,v) \leq \theta(i,w) \leq \theta(i,v) \tag{13}$$

Case 3: If $\theta(i-1,v) = \theta(i-1,w)$ then due to post-condition 21.1 the following is true for all i such that $(i_s+1) \leq i \leq T$.

$$\theta(i-1,v) = \theta(i-1,w) \leq \theta(i,v) \tag{14}$$

SparseNaive Post-condition 23.1 $\forall <v,w> \in E$, *if* $\theta(i,v) < \theta(i,w)$ *then* $<v,w> \in NodeOrd(i)$

Satisfied by the assignment in line 23.

SparseNaive Post-condition 24.1 $\forall q : (i_s+1) \leq q \leq T$ *and* $\forall v \in V$, $\theta(q-1,v) \leq \theta(q,v)$

Satisfied by the loop bounds of the for loop starting at line 15 and post-condition 21.1.

SparseNaive Post-condition 24.2 $\forall q : (i_s+1) \leq q \leq T$ *and* $\forall <v,w> \in E$, $\theta(q-1,v) \leq \theta(q,w)$

Satisfied by the loop bounds of the for loop starting at line 15 and post-condition 22.2.

SparseNaive Post-condition 24.3 $\forall i : (i_s+1) \leq i \leq T$, $NodeOrd(i)$ *is acyclic*

During the upwards tile growth, relations are added to $NodeOrd(i)$ at line 23. Post-condition 14.2 quarantees that $NodeOrd(i_s)$ is initialized as acyclic. Using the inductive assumption that $NodeOrd(i-1)$ is acyclic, we show that each new relation added at line 23 does not cause a cycle with the relations in $NodeOrd(i-1)$ and any other relations previously added during the current execution of line 23.

We assume the contrary and then derive a contradiction. Assume there is a path $<w,x_0> \ldots <x_n,v>$ in the current set of relations such that upon adding $<v,w>$ a cycle would be created. Due to the second inequality in the post-condition 22.1 and line 23 the following statement is true about the tiling function θ values for the nodes in the path at the current convergence iteration i.

$$\theta(i,w) \leq \theta(i,x_0) \leq \cdots \leq \theta(i,x_n) \leq \theta(i,v) \tag{15}$$

Due to line 23, if the relation $<v,w>$ is being added to $NodeOrd(i)$ then the following is true.

$$\theta(i,v) < \theta(i,w) \tag{16}$$

Combining (15) and (16) result in the contradiction that $\theta(i,v) < \theta(i,v)$. Therefore, it is not possible to add a relation $<v,w>$ to $NodeOrd(i)$ which will cause a cycle.

SparseNaive Post-condition 24.4 $\forall i : (i_s+1) \leq i \leq T$ *and* $\forall <v,w>\in E$, *if* $\theta(i,v) < \theta(i,w)$ *then* $<v,w>$ $inNodeOrd(i)$

Satisfied by the loop bounds of the for loop starting at line 15 and post-condition 23.1.

SparseNaive Post-condition 25.1 $\forall i : 1 \leq i \leq (T-1)$ *and* $\forall v \in V$, $\theta(i,v) \leq \theta(i+1,v)$

This condition states that all later convergence iterations performed on the same node x will be in the same or later tile. It depends directly on post-conditions 13.1, and 24.1 which are post-conditions for the downward tile growth, and upward tile growth sections of the algorithm respectively. Between lines 13 and the end of the algorithm, no assignments occur to $\theta(i,x)$ with $1 \leq i \leq i_s$ and $v \in V$. Also, post-condition 24.1 is not invalidated by line 25 in the algorithm.

Notice that upon substitution of $q = i+1$ in post-condition 24.1 we get the following statement.

$$\forall i : (i_s+1) \leq (i+1) \leq T \quad \text{and} \quad \forall v \in V, \quad \theta((i+1)-1,v) \leq \theta(i+1,v) \tag{17}$$

Rewriting (17) we get the following.

$$\forall i : i_s \leq i \leq (T-1) \quad \text{and} \quad \forall v \in V, \quad \theta(i,v) \leq \theta(i+1,v) \tag{18}$$

By combining the domains of i in post-condition 13.1 and (18) we get post-condition 25.1.

SparseNaive Post-condition 25.2 $\forall i : 1 \leq i \leq (T-1)$ *and* $\forall <v,w>\in E$, $\theta(i,v) \leq \theta(i+1,w)$

This condition states that all later convergence iterations performed on the neighbors of any node v will be in the same or later tile. It depends directly on post-conditions 13.2 and 24.2 which are post-conditions for the downward tile growth and upward tile growth sections of the algorithm respectively. Between lines 13 and the end of the algorithm, no assignments occur to $\theta(k,x)$ with $1 \leq i \leq i_s$ and $v \in V$. Also, post-condition 24.2 is not invalidated by line 25 in the algorithm.

Notice that upon substitution of $q = i+1$ in post-condition 24.2 we get the following statement.

$$\forall i : (i_s+1) \leq (i+1) \leq T \quad \text{and} \quad \forall <v,w>\in E, \quad \theta((i+1)-1,v) \leq \theta(i+1,w) \tag{19}$$

Rewriting (19) we get the following.

$$\forall i : i_s \leq i \leq (T-1) \quad \text{and} \quad \forall <v,w>\in E, \quad \theta(i,v) \leq \theta(i+1,w) \tag{20}$$

By combining the domains of i in post-condition 13.2 and (20) we get post-condition 25.2.

SparseNaive Post-condition 25.3 $\forall i : 1 \leq i \leq T, NodeOrd(i)$ *is acyclic*

This condition is satisfed by combining the domains of i in post-conditions 13.3, 14.1, and 24.3 which are post-conditions for the downward tile growth, reinitialization of $NodeOrd(i_s)$, and upward tile growth sections of the algorithm respectively. No assignments are made to the $NodeOrd(i)$ sets in the given ranges between the post-conditions and the end of the program.

SparseNaive Post-condition 25.4 $\forall i : 1 \leq i \leq T$ *and* $\forall <v, w> \in E$, *if* $\theta(i, v) < \theta(i, w)$ *then* $<v, w>$ $in NodeOrd(i)$

This condition is satisfed by combining the domains of i in post-conditions 13.4, 14.2, and 24.4 which are post-conditions for the downward tile growth, reinitialization of $NodeOrd(i_s)$, and upward tile growth sections of the algorithm respectively. No assignments are made to the $NodeOrd(i)$ sets in the given ranges between the post-conditions and the end of the program.

4.3 Reorder the Mesh Nodes

The first step of a typical Gauss-Seidel computation is to assign an arbitrary order σ to the nodes. This affects the result of the computation because at each convergence iteration each mesh node will use the most recent values of the unknowns residing at neighboring nodes in the mesh to update its own unknowns. Mesh nodes at each iteration are computed based on their order.

We have two goals when ordering the nodes: to satisfy the constraints specified in the $NodeOrd$ relation and to increase intra-iteration locality. First and foremost, we must satisfy the $NodeOrd$ relation so that we can show the new execution based on our constructed σ satsifies the Gauss-Seidel Partial Ordering Constraints. Second, want to give consecutive numbers to nodes that at any iteration are executed by the same tile, because the data for a node is stored in memory based on its order. Therefore we want the data associated with nodes executed by the same tile to be close in memory and consequently have better intra-iteration locality. When these two goals conflict, for correctness we always satisfy the $NodeOrd$ relation.

We construct the reordering function σ by performing a topological sort of the nodes based on the $NodeOrd$ relation. SparseNaive post-condition 25.3 (see figure 5) states that $NodeOrd$ is acyclic, so a topological sort is possible. The time complexity of the sort is $O(|V| + |E|)$. The topological sort attempts to give nodes within the same cell of the partitioning consecutive ordering. The resulting σ has the following property:

if $<x, y> \in NodeOrd$ then $\sigma(x) < \sigma(y)$.

4.4 Generate the Sparse Matrix

We generate the matrix using the typical Finite Element Analysis (FEA) matrix assembly functions, except we ensure that rows for the unknowns on each node are consecutive in memory for improved intra-iteration locality. This step is not counted as overhead because it is already part of FEA.

4.5 Reschedule the Sparse Matrix Computation

Inter-iteration locality is improved by executing multiple convergence iterations on a subset of mesh nodes that fit into cache. The typical Gauss-Seidel schedule, as shown in (1), traverses the unknowns associated with all the mesh nodes before moving to the next convergence iteration. Therefore, in order to improve inter-iteration locality we reschedule Gauss-Seidel based on the tiling function θ.

Using the tiling function θ, we generate a set of mesh nodes for each tile to execute at each convergence iteration. We can then use the reordering function σ to translate each set of mesh nodes into the list of matrix rows associated with each of the mesh nodes. As previously described, each matrix row associated with a mesh node will have consecutive ordering. For each mesh node scheduled for a specific tile and convergence iteration $<t, i>$, we add the matrix rows $\{\sigma(v)d, \sigma(v)d+1, ..., \sigma(v)d+(d-1)\}$ to the set $s(t, i)$. The rescheduling step has an upper bound of $O(dT|V|)$, since there are T iteration points for each mesh node and for each on of these iteration points there will be d rows in the sparse matrix, where d is the number of unknowns per mesh node.

4.6 Execute the New Schedule

Finally, we rewrite the Gauss-Seidel computation in (1) as the pseudocode in (21).

```
for $t = 0, ..., (k-1)$
    for $i = 1, 2, ..., T$
        for all $j$ in $sched(t, i)$
            $u_j^{(i)} = (1/a_{jj})(f_j - \sum_{k=1}^{j-1} a_{jk} u_k^{(i)} - \sum_{k=j+1}^{R} a_{jk} u_k^{(i-1)})$          (21)
```

The rewritten Gauss-Seidel computation executes the iteration points tile-by-tile, and within a tile executes one convergence iteration at a time. Within a convergence iteration, matrix rows associated with mesh nodes are executed according to the order given by σ. The complexity of this step is the same as Gauss-Seidel, $O(Td^2(|E|+|V|))$, where $d^2(|E|+|V|)$ is the number of non-zeros in the sparse matrix.

Because of the new schedule in the rewritten Gauss-Seidel computation the following **Execution Properties** hold for the execution function e' based on sparse tiling.

1. for $1 \leq i, q \leq T$ and $\forall v, w \in V$, if $\theta(i, v) < \theta(q, w)$ then $e'(i, v) < e'(q, w)$
   (iteration points in one tile will all be executed before iteration points in later tiles)
2. for $1 \leq i < q \leq T$ and $\forall v, w \in V$, if $\theta(i, v) = \theta(q, w)$ then $e'(i, v) < e'(q, w)$
   (within a tile all iteration points in one convergence iteration are executed before later convergence iterations)
3. for $1 \leq i \leq T$ and $\forall <v, w> \in E$, if $\theta(i, v) = \theta(i, w)$ and $\sigma(v) < \sigma(w)$ then $e'(i, v) < e'(i, w)$
   (neighboring nodes within the same tile maintain the order specified by σ)

Next, we show that the execution e' based on sparse tiling satisfies the Gauss-Seidel Partial Ordering Constraints.

4.7 Proof of Correctness

Given the SparseNaive post-conditions in figures 4 and 5, the property for the reordering function σ given in section 4.3, and the Execution Properties 1-3 of the new schedule given in section 4.6, we show that the sparse tiling based execution e' satisfies the Gauss-Seidel Partial Ordering Constraints.

Gauss-Seidel Partial Ordering Constraint 1 $\forall i : 1 \leq i < (T-1)$ *and* $\forall v \in V$, $e'(i, v) < e'(i+1, v)$.

This partial ordering constraint requires that iterations on any node v are executed in order. Combining SparseNaive post-condition 25.1 and Execution Properties 1 and 2 satisfies this property.

Gauss-Seidel Partial Ordering Constraint 2 *$\forall i : 1 \leq i \leq (T-1)$ and $\forall <v,w> \in E$, if $\sigma(v) < \sigma(w)$ then $e'(i,v) < e'(i,w) < e'(i+1,v)$.*

This partial ordering constraint requires that for all edges $<v,w> \in E$, where node v comes before node w in the node order σ, certain restrictions on the execution of iteration points for v and w must be satisfied. First we will show the following.

$$\forall i : 1 \leq i \leq (T-1) \quad \text{and} \quad <v,w> \in E, \quad \text{if} \quad \sigma(v) < \sigma(w) \quad \text{then} \quad \theta(i,v) \leq \theta(i,w) \leq \theta(i+1,v) \tag{22}$$

For the first inequality of (22), we assume the contrary and derive a contradiction.

$$\exists i : \quad 1 \leq i \leq (T-1) \quad \text{and} \quad <v,w> \in E \quad \text{such that} \quad \sigma(v) < \sigma(w) \quad \text{and} \quad \theta(i,v) > \theta(i,w) \tag{23}$$

Due to SparseNaive post-condition 25.4 and (23) the following statement is true.

$$<w,v> \in NodeOrd(i) \tag{24}$$

Due to (24) and the σ property the following is true.

$$\sigma(w) < \sigma(v) \tag{25}$$

(25) contradicts the assumption in (23), therefore the first inequality of (22) is true.

To obtain the second inequality of (22) we use SparseNaive post-condition 25.2 and the fact that if $<v,w>$ is in E then so is $<w,v>$. Combining (22) with Execution Properties 1 and 3 for e' shows that the sparse Gauss-Seidel Partial Order Property 2 is satisfied.

5 Implementing Sparse Tiling

The **Partition**, **Tile**, **Reorder**, and **Reschedule** steps account for the run-time overhead of sparse tiling. By combining their complexities we get an upper bound of $O(dT|V|+Tk|V||E|)$. Since all of these steps occur at run-time, their efficiency is important. It is possible to rewrite the SparseNaive algorithm using WorkSets to reduce the complexity of the overall overhead to $O(dT|V|+T|E|)$. Figure 7 shows the SparseWorkSet algorithm.

Consider only the downward tile growth phase (the argument for the upward tile growth is similar). In the SparseNaive algorithm the while loop at line 6 is necessary because a specific $\theta(i,v)$ could change multiple times. Such a change occurs if a relation $<v,w>$ is in $NodeOrd(i+1)$ and $\theta(i,w)$ changes due to a relation $<w,z> \in NodeOrd(i+1)$ which is visited later in the foreach loop. The SparseWorkSet algorithm avoids the need for the while loop by incorporating two changes to SparseNaive. First SparseWorkSet has two loops, at lines 5 and 11, over the relations in $NodeOrd(i+1)$. The first loop makes sure that if node v comes before node w in the $NodeOrd$ relation, that the iteration point $<i,w>$ must be in the same or an earlier tile than the iteration point $<i+1,v>$. Since the tiling function θ values for all iteration points $<i+1,v>$

where $v \in V$ won't change at the current i iteration, we only need to visit each $<v, w> \in NodeOrd(i+1)$ once to get $\theta(i, w) \leq \theta(i+1, v)$. The second loop through the relations in $NodeOrd(i+1)$ makes sure that if $<v, w> \in NodeOrd(i+1)$ then iteration point $<i, v>$ is put into the same or earlier tile as iteration point $<i, w>$. Since we visit the relations in $NodeOrd(i+1)$ in order of $\theta(i, w)$, at any node v there won't be a path in $NodeOrd$, $<v, w>, <w, x_1>, ..., <x_{n-1}, x_n>$ where $\theta(i, x_n) < \theta(i, w)$. Therefore, we only need to visit each $<v, w> \in NodeOrd(i+1)$ once in the second foreach loop as well. Upon elimination of the while loop, the complexity of the algorithm changes from $O(Tk|V||E|)$ to $O(T|E|)$.

Another costly part of the SparseNaive algorithm occurs at lines 12 and 23, where each edge $<v, w>$ in the mesh must be checked to determine if $<v, w>$ belongs in $NodeOrd(i)$. If $<v, w>$ isn't in $NodeOrd(i+1)$ it must be the case that $\theta(i+1, v) \geq \theta(i+1, w)$. Thus, we only need to check an edge $<v, w>$ if either $\theta(i, v)$ or $\theta(i, w)$ has changed. In SparseWorkSet, $IterWorkSet(i)$ keeps track of all nodes v whose $\theta(i, v)$ value has changed (gotten smaller). We use it to determine which edges $<v, w> \in E$ must be checked for candidacy in $NodeOrd(i)$. Since the upper bound on the number of edges in $IterWorkSet(i)$ is $|E|$, the worst-case complexity for SparseWorkSet remains $O(T|E|)$.

```
Algorithm SerialSparseWorkset($G(V,E)$,$part$,$T$,$i_s$,$m$)

1:  $\forall v \in V$ and $1 \le i \le T, \theta(i,v) \leftarrow part(v)$
2:  for $1 \le i \le T$, $NodeOrd(i) \leftarrow \emptyset$
3:  $NodeOrd(i_s) \leftarrow \{<v,w> \mid \theta(i_s,v) < \theta(i_s,w)$ and $<v,w> \in E\}$

Downwards tile growth
4:  for $i = i_s - 1$ downto 1
5:      foreach $<v,w> \in NodeOrd(i+1)$
6:          if $\theta(i,w) > \theta(i+1,v)$ then
7:              for $1 \le q \le i$, $\theta(q,w) \leftarrow \theta(i+1,v)$ end for
8:              $w \in IterWorkSet(i)$
9:          end if
10:     end foreach
11:     foreach $<v,w> \in NodeOrd(i+1)$ in order by $\theta(i,w)$
12:         if $\theta(i,v) > \theta(i,w)$ then
13:             for $1 \le q \le i$, $\theta(q,v) \leftarrow \theta(i,w)$ end for
14:             $v \in IterWorkSet(i)$
15:         end if
16:     end foreach
17:     $NodeOrd(i) \leftarrow NodeOrd(i+1)$
18:     foreach $v \in IterWorkSet(i)$; foreach $<v,w> \in E$
19:         if $\theta(i,v) < \theta(i,w)$ then $<v,w> \in NodeOrd(i)$
20:         end if
21:     end foreach; end foreach
22: end for

23: $NodeOrd(i_s) \leftarrow NodeOrd(1)$

Upwards tile growth
24: for $i = i_s + 1$ to $T$
25:     foreach $<v,w> \in NodeOrd(i-1)$
26:         if $\theta(i,v) < \theta(i-1,w)$ then
27:             for $i \le q \le T$, $\theta(q,v) \leftarrow \theta(i-1,w)$ end for
28:             $v \in IterWorkSet(i)$
29:         end if
30:     end foreach
31:     foreach $<v,w> \in NodeOrd(i-1)$ in reverse order of $\theta(i,v)$
32:         if $\theta(i,w) < \theta(i,v)$ then
33:             for $i \le q \le T$, $\theta(q,w) \leftarrow \theta(i,v)$ end for
34:             $w \in IterWorkSet(i)$
35:         end if
36:     end foreach
37:     $NodeOrd(i) \leftarrow NodeOrd(i-1)$
38:     foreach $v \in IterWorkSet(i)$; foreach $<v,w> \in E$
39:             if $\theta(i,v) < \theta(i,w)$ then $<v,w> \in NodeOrd(i)$ end if
40:     end foreach; end foreach

41: $NodeOrd \leftarrow NodeOrd(T)$
```

Figure 7: SparseWorkSet Algorithm

6 Related Work

Related work can be categorized by whether it deals with regular or irregular meshes and whether it attempts to improve intra-iteration locality and/or inter-iteration locality.

Traditional tiling [21, 10, 4, 20, 19, 1, 12] can usually be applied to the loop nest which traverses the unknowns associated with a regular mesh. This is because a regular mesh uses a 2D or 3D array data structure with affine boundaries. These compile-time techniques only work for intra-iteration locality, because the convergence iteration loop is not usually implemented in a way that compilers can recognize as associated with the loop over the sparse matrix. Also, determining the tiling and array padding factors has not been solved for all cases. Rivera and Tseng [15] look more specifically at how to do tiling and array padding for 3D regular meshes.

There has also been work on run-time techniques for improving the intra-iteration locality for irregular meshes [8, 14, 2, 7, 13]. Mitchell et al. [14] describe a compiler optimization which operates on non-affine array references in loops. Sparse matrix data structures require indirect array references, which are a type of non-affine array reference. Also, Im and Yelick [8, 9] describe a code generator called SPARSITY which generates blocked sparse matrix-vector multiply. Both Mitchell and Im's techniques improve spatial and temporal locality on the vectors u and f when dealing with the system $Au = f$. Therefore, when applied to an iterative algorithm such as Gauss-Seidel the intra-iteration locality would improve. However, they do not improve the temporal or inter-iteration locality on the sparse matrix, because in their rescheduled code the entire sparse matrix is traversed each convergence iteration. rescheduling

Increasing inter-iteration locality for iterative computations on regular meshes is explored by [3], [16], [17], and [6].

The only other technique to our knowledge which handles inter-iteration locality for irregular meshes is unstructured cache-blocking by Douglas et al.[3]. They tile the iteration space graph resulting from unstructured grids in the context of the Multigrid algorithm using Gauss-Seidel as a smoother. They achieve overall speedups up to 2 with 2D meshes containing 3983, 15679, and 62207 nodes on an SGI O2. They partition the mesh into cells using the Metis [11] partitioner; the dark lines show the mesh partitioning. They then grow tiles from this partitioning up through the iteration space while respecting the data dependences. Each node needs the most recent values of its neighbors to execute; therefore, the number of nodes they can execute in one tile shrinks each iteration. The resulting tiles are pyramid-shaped. The new schedule for the computation executes all of the tiles atomically, and then does a second phase of computation to deal with the rest of the iteration points.

7 Conclusion

We present an algorithm for generating a sparse tiling for Gauss-Seidel. We also give the full proof showing that a serial execution of sparse tiled Gauss-Seidel is bit-equivalent to standard Gauss-Seidel when both computations start with the same node order. Finally, we present a more efficient version of the sparse tiling algorithm based on WorkSets.

8 Acknowledgements

This work was supported by an AT & T Labs Graduate Research Fellowship and in part by NSF Grant CCR-9808946.

References

[1] Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. *The Journal of Supercomputing*, pages 114–124, November 1992.

[2] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 229–241, Atlanta, Georgia, May 1–4, 1999.

[3] Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiss. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis*, pages 21–40, February 2000.

[4] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[5] Michael R. Garey, David S. Johnson, and L. Stockmeyer. Some simplified *NP*-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[6] Kang Su Gatlin and Larry Carter. Architecture-cognizant divide and conquer algorithms. In *Supercomputing SC'99*, Portland, Oregon, November 1999. ACM Press and IEEE Computer Society Press.

[7] Hwansoo Han and Chau-Wen Tseng. Efficient compiler and run-time support for parallel irregular reductions. *Parallel Computing*, 26(13–14):1861–1887, December 2000.

[8] Eun-Jin Im. *Optimizing the Performance of Sparse Matrix-Vector Multiply.* Ph.d. thesis, University of California, Berkeley, May 2000.

[9] Eun-Jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In V.N.Alexandrov, J.J. Dongarra, and C.J.K.Tan, editors, *Proceedings of the 2001 International Conference on Computational Science*, Lecture Notes in Computer Science, San Francisco, CA, USA, May 28-30, 2001. Springer-Verlag.

[10] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Annual ACM SIGPLAN Symposium on Priniciples of Programming Languages*, pages 319–329, 1988.

[11] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 10 January 1998.

[12] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[13] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 Conference on Supercomputing*, ACM SIGARCH, pages 425–433, N.Y., June 20–25 1999. ACM Press.

[14] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing non-affine array references. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 192–202, Newport Beach, California, October 12–16, 1999. IEEE Computer Society Press.

[15] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In ACM, editor, *SC2000: High Performance Networking and Computing. Dallas Convention Center, Dallas, TX, USA, November 4–10, 2000*, pages 60–61, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. ACM Press and IEEE Computer Society Press.

[16] Sriram Sellappa and Siddhartha Chatterjee. Cache-efficient multigrid algorithms. In V.N.Alexandrov, J.J. Dongarra, and C.J.K.Tan, editors, *Proceedings of the 2001 International Conference on Computational Science*, Lecture Notes in Computer Science, San Francisco, CA, USA, May 28-30, 2001. Springer-Verlag.

[17] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices*, 34(5):215–228, May 1999.

[18] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Rescheduling for locality in sparse matrix computations. In V.N.Alexandrov, J.J. Dongarra, and C.J.K.Tan, editors, *Proceedings of the 2001 International Conference on Computational Science*, Lecture Notes in Computer Science, San Francisco, CA, USA, May 28-30, 2001. Springer-Verlag.

[19] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation*, 1991.

[20] M. Wolfe. More iteration space tiling. In ACM, editor, *Proceedings, Supercomputing '89: November 13–17, 1989, Reno, Nevada*, pages 655–664, New York, NY 10036, USA, 1989. ACM Press.

[21] Michael J. Wolfe. Iteration space tiling for memory hierarchies. In *Parallel Processing for Scientific Computing*, pages 357–361, 1987.