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 #CS2003-0741, April 2003
(Revision of Technical Report #CS2001-0690, December 2001)

Abstract

Gauss-Seidel is an iterative computation used for solving a set of simultaneous linear equations, $A\vec{u} = \vec{f}$. If the matrix A uses a sparse matrix representation, storing only nonzeros, then the data dependences in the computation arise from A's nonzero structure. We use this structure to schedule the computation at runtime using a technique called *full sparse tiling.* The sparse tiled computation exhibits better data locality and therefore improved performance. This paper gives a complete proof that a serial schedule for full sparse tiled Gauss-Seidel generates results equivalent to those that a typical Gauss-Seidel computation produces. We also provide implementation and correctness details for full sparse tiling with reduced worst-case complexity.

1 Introduction

Gauss-Seidel is an iterative method which solves a set of simultaneous linear equations, $A\vec{u} = \vec{f}$. Iterative methods solve for the unknown vector $\vec{u}$ by iteratively traversing the system of equations, converging toward a solution. Pseudo-code for Gauss-Seidel is shown in (1) with u_v representing an element in the vector $\vec{u}$, f_v an element in the vector $\vec{f}$, and a_{vw} the element at row v and column w in the matrix A. We refer to the iterator, *iter*, of the outermost loop as the *convergence iterator*. The v loop iterates over the unknowns and corresponding rows of matrix A. The w loop, which is implicit in the summations, iterates over the columns of matrix A. Upon the completion of each convergence iteration a new value is generated for all the unknowns in $\vec{u}$.

```
for $iter = 1, 2, ..., T$
    for $v = 0, 1, ..., (R-1)$
        $u_v = (1/a_{vv})(f_v - \sum_{w=0}^{v-1} a_{vw}u_w - \sum_{w=v+1}^{(R-1)} a_{vw}u_w)$        (1)
```

If the matrix A is sparse then a compressed format is used to store only the nonzeros in the matrix. Since sparse matrices in typical applications contain less than 1% of the possible entries, the use of compressed formats results in a large storage savings. The loops in computations involving sparse matrices are modified so that the computation visits only the matrix elements actually stored. In (2), we show how the pseudocode for Gauss-Seidel changes to iterate only over the nonzeros within a row of the matrix. Computation only

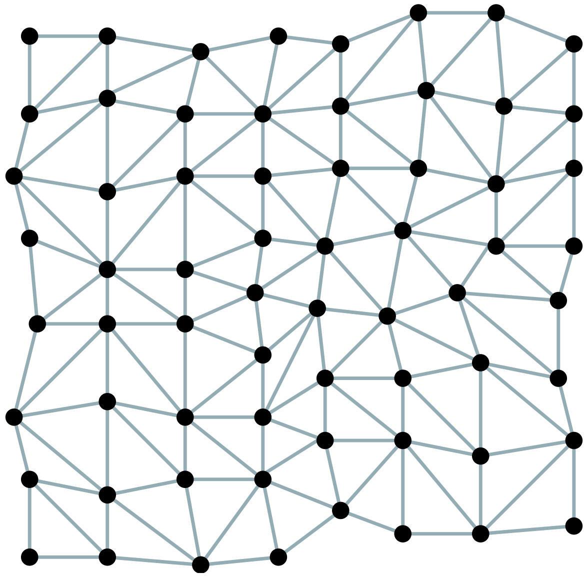

Figure 1: Example Matrix Graph

occurs when there are nonzero matrix entries, therefore compressed formats result in computation savings as well.

```
for $iter = 1, 2, ..., T$
    for $v = 0, 1, ..., (R - 1)$
        $u_v = f_v$
        for all $w$ where $a_{vw} \neq 0$ and $w \neq v$          (2)
            $u_v = u_v - a_{vw} u_w$
        $u_v = u_v / a_{vv}$
```

The nonzero structure in a sparse matrix can be visualized with a matrix graph $G(V, E)$. For each iteration, v, and corresponding unknown, u_v, there is a node in the matrix graph, $v \in V$. There is an directed edge $<v, w> \in E$ if $a_{vw} \neq 0$ or $a_{wv} \neq 0$. Figure 1 shows an example matrix graph with the direction of the edges elided.

The matrix graph for the sparse matrix A induces structure on the Gauss-Seidel iteration space. The iteration space in figure 2 contains three convergence iterations, with an instance of the matrix graph appearing at each convergence iteration. Each iteration point[1] , $<iter, v>$, represents the computations for u_v at convergence iteration $iter$ as specified in (1). The arrows represent data dependences[2] between the iteration points. In Gauss-Seidel, each computation for u_v uses the most recently calculated values of the neighboring unknowns in the matrix graph, therefore some data dependences come from iteration points in the same convergence iteration, $iter$, and some data dependences come from points in the previous convergence iteration, $iter - 1$.

The typical schedule for Gauss-Seidel, as shown in (2), is to complete all the computation for each convergence iteration $iter$ before doing the next convergence iteration. Sparse tiling techniques reschedule the computation so that subsets of unknowns are computed across multiple convergence iterations. Figure 3

[1] We use the term iteration point for points in the iteration space graph and node for points in the matrix graph.

[2] Only the dependences for one matrix node are shown for clarity.

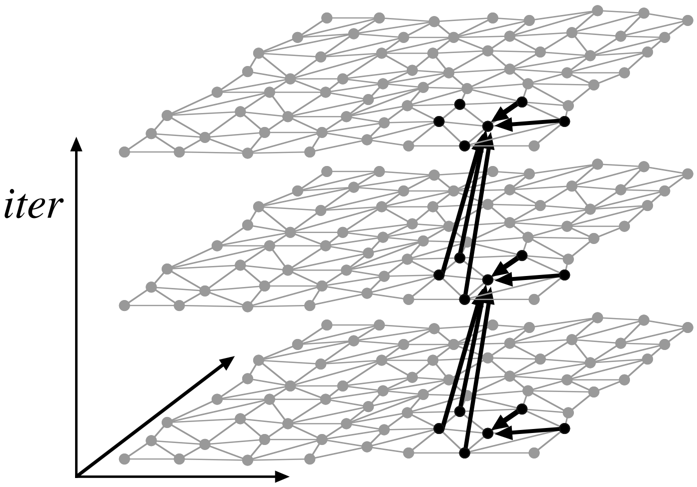

Figure 2: Gauss-Seidel iteration space graph for 3 convergence iterations.

exhibits a full sparse tiling of the Gauss-Seidel iteration space in figure 2. The code is transformed into an inspector/executor framework. The inspector generates sparse tiles the iteration space based on the matrix graph and creates a new schedule based on the sparse tiled computation. The executor is the original computation code transformed so that it uses the schedule generated by the inspector. Sparse tiled Gauss-Seidel results in improved data locality [21]. It is possible to determine a parallel schedule for the full sparse tiled computation [22], but this paper focuses on the legality of a serial schedule.

Section 2 will describe how Gauss-Seidel for the compressed sparse row (CSR) format is transformed at compile time to use a run-time generated sparse tiling schedule. It also includes an in-depth analysis of the data dependences for Gauss-Seidel written for CSR. Theorem 1 specifies the constraints the sparse tiling function must satisfy so that a serial tile-by-tile schedule generates results equivalent to the typical Gauss-Seidel schedule. The construction of the sparse tiling function is detailed in section 3. Finally, in section 3.6 the properties of the tiling function generated as described in section 3 are used to show that the constraints in theorem 1 are satisfied; therefore, full sparse tiled Gauss-Seidel for CSR generates equivalent results to typical Gauss-Seidel for CSR when they both start with the same data ordering function.

2 Transforming the Code

This paper proves that the transformed code shown in figure 6 generates an unknown array $\mathbf{u}'$, equivalent to the results from the Gauss-Seidel code shown in figure 5 when both versions of the code use the same data ordering function.To prove the correctness of the sparse tiled Gauss-Seidel in figure 6, we first determine the data dependence relations for the Gauss-Seidel pseudocode shown in figure 5. We then specify the compile-time mappings [14] which transform Gauss-Seidel for CSR to sparse tiled Gauss-Seidel for CSR. Finally, we determine the constraints which the tiling function and data ordering generated by sparse tiling techniques must satisfy to make the transformation legal. A transformation is legal if all the data dependences are satisfied in the transformed iteration space.

The original order, $v = 0, 1, ..., (R - 1)$, given to the unknowns and corresponding matrix rows is often

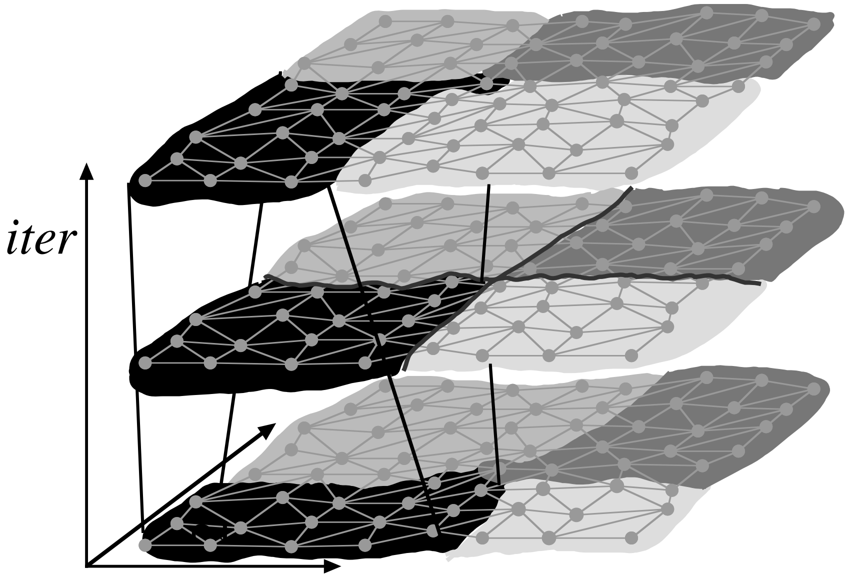

Figure 3: Full sparse tiled Gauss-Seidel iteration space

arbitrary and may be changed without affecting the convergence properties of Gauss-Seidel [26]. Therefore, if the unknowns are mapped to another order before performing Gauss-Seidel, the final result will be somewhat different, but the Gauss-Seidel convergence properties still hold. Since full sparse tiling (and cache blocking) perform an initial reordering of the unknowns, to prove correctness of sparse tiled Gauss-Seidel we will compare the result of full sparse tiled Gauss-Seidel with the generated reordering, $i = \sigma(v)$ where i is the loop iterator in Figures 5 and 6, to that of typical Gauss-Seidel using the same initial ordering. Therefore, in Figures 5 and 6, assume that the original matrix A, unknown vector $\vec{u}$, and right-hand side $\vec{f}$ have been reordered using the reordering function $i = \sigma(v)$ such that $A'_{\sigma(v)\sigma(w)} = A_{vw}$, $u'_{\sigma(v)} = u_v$, and $f'_{\sigma(v)} = f_v$.

Figure 5 gives detailed pseudocode for Gauss-Seidel written for the Compresses Sparse Row (CSR) matrix format, which stores the nonzeros in a sparse matrix A' by row using the **ia**, **ja**, and **a** arrays (see Figure 4). The values in the **ia** array index into the **ja** and **a** arrays indicating where the column identities and the nonzero matrix elements start for row i. The vectors $\vec{u'}$ and $\vec{f'}$ are represented with arrays $\mathbf{u}'$ and $\mathbf{f}'$.

At runtime, sparse tiling techniques generate a data ordering function σ and a *tiling function*, $\theta(iter, v) : \{1, .., T\}$ x $V \rightarrow \{0, ..., (k-1)\}$. The tiling function assigns iteration points, $< iter, v >$, from the original iteration space to tiles. From this tiling function a *schedule function* is created. The schedule function, $sched(tileID, iter) : \{0, ..., (k-1)\}$ x $\{1, ..., T\} \rightarrow 2^{\{0,...,(R-1)\}}$, specifies for each tile and convergence iteration which subset of the reordered unknowns must be updated. The transformed code shown in figure 6 does a tile-by-tile execution of the iteration points by using the schedule function, which is defined as $sched(tileID, iter) = \{\sigma(v) \mid \theta(iter, v) = tileID\}$.

2.1 Data Dependence Relations in Gauss-Seidel for CSR

Each statement exists within an iteration space defined by the surrounding loops. For example, statement 1 in Figure 5 lies within the iteration space $\{[iter, i] \mid (1 \leq iter \leq T)$ and $(0 \leq i < R)\}$. Data dependences indicate when an instance of a statement must execute before the instance of another statement or before another instance of the same statement due to memory-based data dependences. Data dependence relations are sets of mappings between statement instances [14]. For example, since statement 1 in Figure 5 writes to

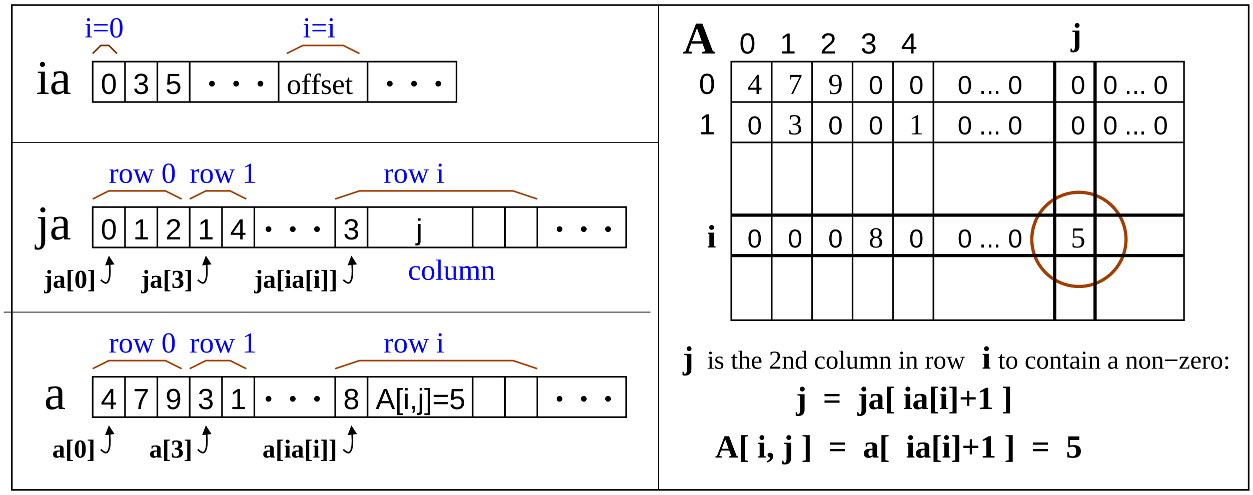

Figure 4: Compressed sparse row (CSR) format

```
GaussSeidelCSR($A'$(ia,ja,a),u',f')
      for $iter = 1, T$ do
            for $i = 0, (R-1)$ do
1:                u'$[i]$ = f'$[i]$
                  for $p$=ia$[i]$, ia$[i+1]-1$ do
                        if ( ja$[p] \neq i$ ) then
2:                            u'$[i]$ = u'$[i]$ - a$[p]$ * u'[ja$[p]$]
                        else
3:                            diag$[i]$ = a$[p]$
                        endif
                  endfor
4:                u'$[i]$ = u'$[i]$/diag$[i]$
            endfor
      endfor
```

Figure 5: Gauss-Seidel for Compressed Sparse Row (CSR)

the same memory location at each convergence iterations *iter*, there is a memory-based dependence which can be specified with the following data dependence relation.

$$\{[iter_1, i] \rightarrow [iter_2, i] \mid 1 \leq iter_1 < iter_2 \leq T \text{ and } 0 \leq i < R\}$$

By using uninterpreted function symbols, it is possible to represent the data dependence relations between indirect memory references as well. The values in the index arrays **ia** and **ja** are not known until runtime, therefore we represent those values abstractly with the uninterpreted function symbols $ia()$ and $ja()$. For example, the dependence relation between the write of $\mathbf{u}'[i]$ in statement 1 of Figure 5 and the read of $\mathbf{u}'[\mathbf{ja}[p]]$ in statement 3 is as follows.

$$\{[iter_1, i] \rightarrow [iter_2, i, p] \mid 1 \leq iter_1 \leq iter_2 \leq T \text{ and } 0 \leq i < R \text{ and } ia(i) \leq p < ia(i+1) \text{ and } i = ja(p)\}$$

In order to perform Gauss-Seidel on the matrix A' there must be nonzero values on the diagonals. Therefore,

```
GaussSeidelCSR_ST($A'(\mathbf{ia},\mathbf{ja},\mathbf{a}),\mathbf{u}',\mathbf{f}',sched,k$)
    for $tileID = 0, (k-1)$ do
        for $iter = 1, T$ do
            for $i \in sched(tileID, iter)$
1:              $\mathbf{u}'[i] = \mathbf{f}'[i]$
                for $p$=$\mathbf{ia}[i]$,$\mathbf{ia}[i+1]-1$ do
                    if ( $\mathbf{ja}[p] \neq i$ ) then
2:                      $\mathbf{u}'[i] = \mathbf{u}'[i]$ - $\mathbf{a}[p]$ * $\mathbf{u}'[\mathbf{ja}[p]]$
                    else
3:                      $\mathbf{diag}[i] = \mathbf{a}[p]$
                    endif
                endfor
4:              $\mathbf{u}'[i] = \mathbf{u}'[i]/\mathbf{diag}[i]$
            endfor
        endfor
    endfor
```

Figure 6: Serial execution of sparse tiled Gauss-Seidel for Compressed Sparse Row (CSR)

we assume that the array variable **diag** in Figure 5 is assigned at least once for each iteration of the i loop. Since it is assigned and then used, it will not affect the legality of reordering any of the loops.

Tables 2.2 and 2.2 list all of the data dependence relations for the code in figure 5.

2.2 Compile-time Transformation Mappings

The sparse tiled code in Figure 6 executes points in the iteration space $\{[iter, i] \mid (1 \leq iter \leq T)$ and $(0 \leq i < R)\}$ in a tile-by-tile fashion. The following integer space mappings[14] describe the code transformation from the code in Figure 5 to that in Figure 6. Transformation mapping M_s maps statement s from its original iteration space to a unified iteration space.

$$
\begin{aligned}
M_1 = \{[iter, i] &\quad\rightarrow\quad [\theta(iter, \sigma^{-1}(i)), iter, i, 1, 1, 1]\} && (3)\\
M_2 = \{[iter, i, p] &\quad\rightarrow\quad [\theta(iter, \sigma^{-1}(i)), iter, i, 2, p, 1]\} && (4)\\
M_3 = \{[iter, i, p] &\quad\rightarrow\quad [\theta(iter, \sigma^{-1}(i)), iter, i, 2, p, 2\} && (5)\\
M_4 = \{[iter, i] &\quad\rightarrow\quad [\theta(iter, \sigma^{-1}(i)), iter, i, 3, 1, 1]\} && (6)
\end{aligned}
$$

The unified iteration space will be executed in lexicographic order. Lexicographical order on integer tuples may be defined as follows [13]:

$$(x_1, ..., x_n) \prec (y_1, ..., y_n) \Leftrightarrow \exists m: \ (\forall i: \ 1 \leq i < m \Rightarrow x_i = y_i) \wedge (x_m < y_m)$$

Therefore, according to the transformation mappings M_1, M_2, M_3, and M_4, the unified iteration space is executed by tile, by convergence iteration, and then by the order specified with the σ function since $i = \sigma(v)$.

ID	Ref 1	Ref 2	Data dependence relation
D1	1: $\mathbf{u}'[i]$	1: $\mathbf{u}'[i]$	$\{[iter_1, i] \rightarrow [iter_2, i] \mid 1 \leq iter_1 < iter_2 \leq T$ and $0 \leq i < R\}$
D2	1: $\mathbf{u}'[i]$	2: $\mathbf{u}'[i]$ (read)	$\{[iter_1, i] \rightarrow [iter_2, i, p] \mid 1 \leq iter_1 \leq iter_2 \leq T$ and $0 \leq i < R$ and $ia(i) \leq p < ia(i+1)$ and $i \neq ja(p)\}$
D3	1: $\mathbf{u}'[i]$	2: $\mathbf{u}'[i]$ (write)	same as dependence 2
D4	1: $\mathbf{u}'[i]$	2: $\mathbf{u}'[\mathbf{ja}[p]]$	$\{[iter, i_1] \rightarrow [iter, i_2, p] \mid 1 \leq iter \leq T$ and $0 \leq i_1 < i_2 < R$ and $ia(i_2) \leq p < ia(i_2+1)$ and $i_1 = ja(p)\}$ union $\{[iter_1, i_1] \rightarrow [iter_2, i_2, p] \mid 1 \leq iter_1 < iter_2 \leq T$ and $0 \leq i_1, i_2 < R$ and $i_1 \neq i_2$ and $ia(i_2) \leq p < ia(i_2+1)$ and $i_1 = ja(p)\}$
D5	1: $\mathbf{u}'[i]$	3: $\mathbf{u}'[\mathbf{ja}[p]]$	$\{[iter_1, i] \rightarrow [iter_2, i, p] \mid 1 \leq iter_1 \leq iter_2 \leq T$ and $0 \leq i < R$ and $ia(i) \leq p < ia(i+1)$ and $i = ja(p)\}$
D6	1: $\mathbf{u}'[i]$	4: $\mathbf{u}'[i]$ (read)	$\{[iter_1, i] \rightarrow [iter_2, i] \mid 1 \leq iter_1 \leq iter_2 \leq T$ and $0 \leq i < R\}$
D7	1: $\mathbf{u}'[i]$	4: $\mathbf{u}'[i]$ (write)	same as dependence 6
D8	2: $\mathbf{u}'[i]$ (read)	1: $\mathbf{u}'[i]$ (read)	read-read dependence
D9	2: $\mathbf{u}'[i]$ (read)	2: $\mathbf{u}'[i]$ (read)	read-read dependence
D10	2: $\mathbf{u}'[i]$ (read)	2: $\mathbf{u}'[i]$ (write)	$\{[iter, i, p_1] \rightarrow [iter, i, p_2] \mid 1 \leq iter \leq T$ and $0 \leq i < R$ and $p_1 \leq p_2$ and $ia(i) \leq p_1, p_2 < ia(i+1)$ and $i \neq ja(p_1)$ and $i \neq ja(p_2)\}$ union $\{[iter_1, i, p_1] \rightarrow [iter_2, i, p_2] \mid 1 \leq iter_1 < iter_2 \leq T$ and $0 \leq i < R$ and $ia(i) \leq p_1, p_2 < ia(i+1)$ and $i \neq ja(p_1)$ and $i \neq ja(p_2)\}$
D11	2: $\mathbf{u}'[i]$ (read)	2: $\mathbf{u}'[\mathbf{ja}[p]]$	read-read dependence
D12	2: $\mathbf{u}'[i]$ (read)	3: $\mathbf{u}'[\mathbf{ja}[p]]$	read-read dependence
D13	2: $\mathbf{u}'[i]$ (read)	4: $\mathbf{u}'[i]$ (read)	read-read dependence
D14	2: $\mathbf{u}'[i]$ (read)	4: $\mathbf{u}'[i]$ (write)	$\{[iter_1, i, p] \rightarrow [iter_2, i] \mid 1 \leq iter_1 \leq iter_2 \leq T$ and $0 \leq i < R$ and $ia(i) \leq p < ia(i+1)$ and $i \neq ja(p)\}$
D15	2: $\mathbf{u}'[i]$ (write)	1: $\mathbf{u}'[i]$ (read)	$\{[iter_1, i, p] \rightarrow [iter_2, i] \mid 1 \leq iter_1 < iter_2 \leq T$ and $0 \leq i < R$ and $ia(i) \leq p < ia(i+1)$ and $i \neq ja(p)\}$
D16	2: $\mathbf{u}'[i]$ (write)	2: $\mathbf{u}'[i]$ (read)	$\{[iter, i, p_1] \rightarrow [iter, i, p_2] \mid 1 \leq iter \leq T$ and $0 \leq i < R$ and $p_1 < p_2$ and $ia(i) \leq p_1, p_2 < ia(i+1)$ and $i \neq ja(p_1)$ and $i \neq ja(p_2)\}$ union $\{[iter_1, i, p_1] \rightarrow [iter_2, i, p_2] \mid 1 \leq iter_1 < iter_2 \leq T$ and $0 \leq i < R$ and $ia(i) \leq p_1, p_2 < ia(i+1)$ and $i \neq ja(p_1)$ and $i \neq ja(p_2)\}$
D17	2: $\mathbf{u}'[i]$ (write)	2: $\mathbf{u}'[i]$ (write)	same as dependence 16
D18	2: $\mathbf{u}'[i]$ (write)	2: $\mathbf{u}'[\mathbf{ja}[p]]$	$\{[iter, i_1, p_1] \rightarrow [iter, i_2, p_2] \mid 1 \leq iter \leq T$ and $0 \leq i_1 < i_2 < R$ and $ia(i_1) \leq p_1 < ia(i_1+1)$ and $i_1 \neq ja(p_1)$ and $ia(i_2) \leq p_2 < ia(i_2+1)$ and $i_2 \neq ja(p_2)$ and $i_1 = ja(p_2)\}$ union $\{[iter_1, i_1, p_1] \rightarrow [iter_2, i_2, p_2] \mid 1 \leq iter_1 < iter_2 \leq T$ and $0 \leq i_1, i_2 < R$ and $ia(i_1) \leq p_1 < ia(i_1+1)$ and $i_1 \neq ja(p_1)$ and $ia(i_2) \leq p_2 < ia(i_2+1)$ and $i_2 \neq ja(p_2)$ and $i_1 = ja(p_2)\}$
D19	2: $\mathbf{u}'[i]$ (write)	3: $\mathbf{u}'[\mathbf{ja}[p]]$	$\{[iter, i, p_1] \rightarrow [iter, i, p_2] \mid 1 \leq iter \leq T$ and $0 \leq i < R$ and $ia(i) \leq p_1 < p_2 < ia(i+1)$ and $i \neq ja(p_1)$ and $i = ja(p_2)\}$ union $\{[iter_1, i, p_1] \rightarrow [iter_2, i, p_2] \mid 1 \leq iter_1 < iter_2 \leq T$ and $0 \leq i < R$ and $ia(i) \leq p_1, p_2 < ia(i+1)$ and $i \neq ja(p_1)$ and $i = ja(p_2)\}$
D20	2: $\mathbf{u}'[i]$ (write)	4: $\mathbf{u}'[i]$ (read)	same as dependence 14
D21	2: $\mathbf{u}'[i]$ (write)	4: $\mathbf{u}'[i]$ (write)	same as dependence 14

ID	Ref 1	Ref 2	Data dependence relation
D22	2: $\mathbf{u}'[\mathbf{ja}[p]]$	1: $\mathbf{u}'[i]$	$\{[iter, i_1, p] \to [iter, i_2] \mid 1 \le iter \le T \text{ and } 0 \le i_1 < i_2 < R \text{ and } ia(i_1) \le p < ia(i_1+1) \text{ and } i_1 \ne ja(p) \text{ and } ja(p) = i_2\}$ union $\{[iter_1, i_1, p] \to [iter_2, i_2] \mid 1 \le iter_1 < iter_2 \le T \text{ and } 0 \le i_1, i_2 < R \text{ and } ia(i_1) \le p < ia(i_1+1) \text{ and } i_1 \ne ja(p) \text{ and } ja(p) = i_2\}$
D23	2: $\mathbf{u}'[\mathbf{ja}[p]]$	2: $\mathbf{u}'[i]$ (read)	read-read dependence
D24	2: $\mathbf{u}'[\mathbf{ja}[p]]$	2: $\mathbf{u}'[i]$ (write)	$\{[iter, i_1, p_1] \to [iter, i_2, p_2] \mid 1 \le iter \le T \text{ and } 0 \le i_1 < i_2 < R \text{ and } ia(i_1) \le p_1 < ia(i_1+1) \text{ and } i_1 \ne ja(p_1) \text{ and } ia(i_2) \le p_2 < ia(i_2+1) \text{ and } i_2 \ne ja(p_2) \text{ and } ja(p_1) = i_2\}$ union $\{[iter_1, i_1, p_1] \to [iter_2, i_2, p_2] \mid 1 \le iter_1 < iter_2 \le T \text{ and } 0 \le i_1, i_2 < R \text{ and } ia(i_1) \le p_1 < ia(i_1+1) \text{ and } i_1 \ne ja(p_1) \text{ and } ia(i_2) \le p_2 < ia(i_2+1) \text{ and } i_2 \ne ja(p_2) \text{ and } ja(p_1) = i_2\}$
D25	2: $\mathbf{u}'[\mathbf{ja}[p]]$	2: $\mathbf{u}'[\mathbf{ja}[p]]$	read-read dependence
D26	2: $\mathbf{u}'[\mathbf{ja}[p]]$	3: $\mathbf{u}'[\mathbf{ja}[p]]$	read-read dependence
D27	2: $\mathbf{u}'[\mathbf{ja}[p]]$	4: $\mathbf{u}'[i]$ (read)	read-read dependence
D28	2: $\mathbf{u}'[\mathbf{ja}[p]]$	4: $\mathbf{u}'[i]$ (write)	same as dependence 22
D29	3: $\mathbf{u}'[\mathbf{ja}[p]]$	1: $\mathbf{u}'[i]$	$\{[iter_1, i, p] \to [iter_2, i] \mid 1 \le iter_1 < iter_2 \le T \text{ and } 0 \le i < R \text{ and } ia(i) \le p < ia(i+1) \text{ and } i = ja(p)\}$
D30	3: $\mathbf{u}'[\mathbf{ja}[p]]$	2: $\mathbf{u}'[i]$ (read)	read-read dependence
D31	3: $\mathbf{u}'[\mathbf{ja}[p]]$	2: $\mathbf{u}'[i]$ (write)	$\{[iter, i, p_1] \to [iter, i, p_2] \mid 1 \le iter \le T \text{ and } 0 \le i < R \text{ and } ia(i) \le p_1 < p_2 < ia(i+1) \text{ and } i \ne ja(p_2) \text{ and } ja(p_1) = i\}$ union $\{[iter_1, i, p_1] \to [iter_2, i, p_2] \mid 1 \le iter_1 < iter_2 \le T \text{ and } 0 \le i < R \text{ and } ia(i) \le p_1, p_2 < ia(i+1) \text{ and } i \ne ja(p_2) \text{ and } ja(p_1) = i\}$
D32	3: $\mathbf{u}'[\mathbf{ja}[p]]$	2: $\mathbf{u}'[\mathbf{ja}[p]]$	read-read dependence
D33	3: $\mathbf{u}'[\mathbf{ja}[p]]$	3: $\mathbf{u}'[\mathbf{ja}[p]]$	read-read dependence
D34	3: $\mathbf{u}'[\mathbf{ja}[p]]$	4: $\mathbf{u}'[i]$ (read)	read-read dependence
D35	3: $\mathbf{u}'[\mathbf{ja}[p]]$	4: $\mathbf{u}'[i]$ (write)	$\{[iter_1, i, p] \to [iter_2, i] \mid 1 \le iter_1 \le iter_2 \le T \text{ and } 0 \le i < R \text{ and } ia(i) \le p < ia(i+1) \text{ and } i = ja(p)\}$
D36	4: $\mathbf{u}'[i]$ (read)	1: $\mathbf{u}'[i]$	same as dependence 1
D37	4: $\mathbf{u}'[i]$ (read)	2: $\mathbf{u}'[i]$ (read)	read-read dependence
D38	4: $\mathbf{u}'[i]$ (read)	2: $\mathbf{u}'[i]$ (write)	$\{[iter_1, i] \to [iter_2, i, p] \mid 1 \le iter_1 < iter_2 \le T \text{ and } 0 \le i < R \text{ and } ia(i) \le p < ia(i+1) \text{ and } i \ne ja(p)\}$
D39	4: $\mathbf{u}'[i]$ (read)	2: $\mathbf{u}'[\mathbf{ja}[p]]$	read-read dependence
D40	4: $\mathbf{u}'[i]$ (read)	3: $\mathbf{u}'[\mathbf{ja}[p]]$	read-read dependence
D41	4: $\mathbf{u}'[i]$ (read)	4: $\mathbf{u}'[i]$ (read)	read-read dependence
D42	4: $\mathbf{u}'[i]$ (read)	4: $\mathbf{u}'[i]$ (write)	same as dependence 6
D43	4: $\mathbf{u}'[i]$ (write)	1: $\mathbf{u}'[i]$	same as dependence 1
D44	4: $\mathbf{u}'[i]$ (write)	2: $\mathbf{u}'[i]$ (read)	same as dependence 38
D45	4: $\mathbf{u}'[i]$ (write)	2: $\mathbf{u}'[i]$ (write)	same as dependence 38
D46	4: $\mathbf{u}'[i]$ (write)	2: $\mathbf{u}'[\mathbf{ja}[p]]$	same as dependence 4
D47	4: $\mathbf{u}'[i]$ (write)	3: $\mathbf{u}'[\mathbf{ja}[p]]$	$\{[iter_1, i] \to [iter_2, i, p] \mid 1 \le iter_1 < iter_2 \le T \text{ and } 0 \le i < R \text{ and } ia(i) \le p < ia(i+1) \text{ and } i = ja(p)\}$
D48	4: $\mathbf{u}'[i]$ (write)	4: $\mathbf{u}'[i]$ (read)	same as dependence 1
D49	4: $\mathbf{u}'[i]$ (write)	4: $\mathbf{u}'[i]$ (write)	same as dependence 1

2.3 Constraints on the Tiling Function Due to Data Dependences

In this section we use the data dependence relations for the code in Figure 5 and the transformation mappings which transform the code in Figure 5 to the code in Figure 6 to derive the necessary constraints on the tiling function θ and reordering function σ. The legality of transformation mappings are verified by applying the transformation mappings to the statements involved in each data dependence relation and verifying that the data dependence remains satisfied. If there is a dependence between iteration $\vec{x}$ of statement s and iteration $\vec{y}$ of statement q then the code transformation mappings must satisfy that dependence [14]. Specifically the following constraints must be satisfied,

$$\forall \vec{x}, \vec{y}, s, q, R, T : \vec{x} \rightarrow \vec{y} \in d_{sq} \Rightarrow M_s(\vec{x}) \prec M_q(\vec{y})$$

where R and T are the symbolic constants in Gauss-Seidel for CSR, and $\prec$ is the lexicographic ordering operator.

For example, the dependences between statement 1 and itself are as follows.

$$d_{11} = \{[iter_1, i] \rightarrow [iter_2, i] \mid (1 \leq iter_1 < iter_2 \leq T) \wedge (0 \leq i < R)\}$$

Applying M_1 to both sides of the data dependence relation results in the following constraint.

$$\begin{aligned}&\forall\; iter_1, iter_2, i : (1 \leq iter_1 < iter_2 \leq T) \wedge (0 \leq i < R)\\ &\Rightarrow [\theta(iter_1, \sigma^{-1}(i)), iter_1, i, 1, 1, 1] \prec [\theta(iter_2, \sigma^{-1}(i)), iter_2, i, 1, 1, 1]\end{aligned}$$

By using the definition of lexicographical order, the above constraint simplifies to the following.

$$\forall\; iter_1, iter_2, i : (1 \leq iter_1 < iter_2 \leq T) \wedge (0 \leq i < R) \Rightarrow \theta(iter_1, \sigma^{-1}(i)) \leq \theta(iter_2, \sigma^{-1}(i)) \qquad (7)$$

The dependences going from statement 1 to statement 2 provide a more complex example.

$$\begin{array}{llll} d_{12} = & D2 \cup D4 & & \\ = & \{[iter_1, i] \rightarrow [iter_2, i, p] \mid & (1 \leq iter_1 \leq iter_2 \leq T) & \\ & & \wedge(0 \leq i < R) \wedge (ia(i) \leq p < ia(i+1)) \wedge (i \neq ja(p)) & \\ \cup & \{[iter, i_1] \rightarrow [iter, i_2, p] \mid & (1 \leq iter \leq T) & \\ & & \wedge\ (0 \leq i_1 < i_2 < R) \wedge (ia(i_2) \leq p < ia(i_2+1)) \wedge (i_1 = ja(p))\} & \\ \cup & \{[iter_1, i_1] \rightarrow [iter_2, i_2, p] \mid & (1 \leq iter_1 < iter_2 \leq T) & \\ & & \wedge\ (0 \leq i_1, i_2 < R) \wedge (i_1 \neq i_2) \wedge (ia(i_2) \leq p < ia(i_2+1)) \wedge (i_1 = ja(p))\} & \end{array}$$

Applying M_1 to the left-hand side of the dependence relations and M_2 to the right-hand side results in the following three constraints.

First:

$$\begin{array}{ll} \forall\; iter_1, iter_2, i : & (1 \leq iter_1 < iter_2 \leq T) \wedge (0 \leq i < R) \\ & \wedge(\exists p : ia(i) \leq p < ia(i+1)) \wedge (i \neq ja(p)) \\ & \Rightarrow [\theta(iter_1, \sigma^{-1}(i)), iter_1, i, 1, 1, 1] \prec [\theta(iter_2, \sigma^{-1}(i)), iter_2, i, 2, p, 1] \end{array}$$

By applying the definition of lexicographical order, the above simplifies to the following, which is subsumed by the previously determined constraint (7).

$$\begin{array}{ll} \forall\; iter_1, iter_2, i : & (1 \leq iter_1 < iter_2 \leq T) \wedge (0 \leq i < R) \\ & \wedge(\exists p : ia(i) \leq p < ia(i+1)) \wedge (i \neq ja(p)) \\ & \Rightarrow \theta(iter_1, \sigma(i)^{-1}) \leq \theta(iter_2, \sigma(i)^{-1}) \end{array}$$

Second:

$$\begin{aligned}\forall\, iter, i_1, i_2 : \quad & (1 \le iter \le T) \wedge (0 \le i_1 < i_2 < R) \\ & \wedge(\exists p : ia(i_2) \le p < ia(i_2+1)) \wedge (i_1 = ja(p)) \\ & \Rightarrow [\theta(iter, \sigma^{-1}(i_1)), iter, i_1, 1, 1, 1] \prec [\theta(iter, \sigma^{-1}(i_2)), iter, i_2, 2, p, 1]\end{aligned}$$

By applying the definition of lexicographical order, the above simplifies to the following.

$$\begin{aligned}\forall\, iter, i_1, i_2 : \ & (1 \le iter \le T) \wedge (0 \le i_1 < i_2 < R) \\ & \wedge(\exists p : ia(i_2) \le p < ia(i_2+1)) \wedge (i_1 = ja(p)) \\ & \Rightarrow \theta(iter, \sigma(i_1)^{-1}) \le \theta(iter, \sigma(i_2)^{-1})\end{aligned} \tag{8}$$

Third:

$$\begin{aligned}\forall\, iter_1, iter_2, i_1, i_2 : \quad & (1 \le iter_1 < iter_2 \le T) \wedge (0 \le i_1, i_2 < R) \wedge (i_1 \ne i_2) \\ & \wedge(\exists p : ia(i_2) \le p < ia(i_2+1)) \wedge (i_1 = ja(p)) \\ & \Rightarrow [\theta(iter_1, \sigma^{-1}(i_1)), iter_1, i_1, 1, 1, 1] \prec [\theta(iter_2, \sigma^{-1}(i_2)), iter_2, i_2, 2, p, 1]\end{aligned}$$

By applying the definition of lexicographical order, the above simplifies to the following.

$$\begin{aligned}\forall\, iter_1, iter_2, i_1, i_2 : \ & (1 \le iter_1 < iter_2 \le T) \wedge (0 \le i_1, i_2 < R) \wedge (i_1 \ne i_2) \\ & \wedge(\exists p : ia(i_2) \le p < ia(i_2+1)) \wedge (i_1 = ja(p)) \\ & \Rightarrow \theta(iter_1, \sigma(i_1)^{-1}) \le \theta(iter_2, \sigma(i_2)^{-1})\end{aligned} \tag{9}$$

By completing the process of applying the transformation mappings to the data dependences and simplifying the resulting constraints, it is possible to generate the constraints which must be satisfied by the tiling function θ and the reordering function σ.

Theorem 1 *For a given ordering function σ such that $i = \sigma(v)$ and tiling function θ with the schedule function $sched(tileID, iter) = \{\sigma(v) \mid \theta(iter, v) = tileID\}$, the sparse tiled Gauss-Seidel pseudocode for CSR shown in figure 6 satisfies the Gauss-Seidel for CSR data dependences if the following constraints are met.*

1. $\forall\, iter_1, iter_2, i : (1 \le iter_1 < iter_2 \le T) \wedge (0 \le i < R) \Rightarrow \theta(iter_1, \sigma^{-1}(i)) \le \theta(iter_2, \sigma^{-1}(i))$
2. $\forall\, iter, i_1, i_2 : (1 \le iter \le T) \wedge (0 \le i_1 < i_2 < R) \wedge (\exists p : ia(i_1) \le p < ia(i_1+1) \wedge i_2 = ja(p)) \Rightarrow \theta(iter, \sigma^{-1}(i_1)) \le \theta(iter, \sigma^{-1}(i_2))$
3. $\forall\, iter, i_1, i_2 : (1 \le iter \le T) \wedge (0 \le i_1 < i_2 < R) \wedge (\exists p : ia(i_2) \le p < ia(i_2+1) \wedge i_1 = ja(p)) \Rightarrow \theta(iter, \sigma^{-1}(i_1)) \le \theta(iter, \sigma^{-1}(i_2))$
4. $\forall\, iter_1, iter_2, i_1, i_2 : (1 \le iter_1 < iter_2 \le T) \wedge (0 \le i_1 \ne i_2 < R) \wedge (\exists p : ia(i_1) \le p < ia(i_1+1) \wedge i_2 = ja(p)) \Rightarrow \theta(iter_1, \sigma^{-1}(i_1)) \le \theta(iter_2, \sigma^{-1}(i_2))$
5. $\forall\, iter_1, iter_2, i_1, i_2 : (1 \le iter_1 < iter_2 \le T) \wedge (0 \le i_1 \ne i_2 < R) \wedge (\exists p : ia(i_2) \le p < ia(i_2+1) \wedge i_1 = ja(p)) \Rightarrow \theta(iter_1, \sigma^{-1}(i_1)) \le \theta(iter_2, \sigma^{-1}(i_2))$

Proof: The constraints are generated by testing the legality of the mappings from Gauss-Seidel to sparse tiled Gauss-Seidel. For example, the derivation of constraint 1 in the theorem is the same as that of (7), constraint 3 follows from equation (8), and constraint 5 follows from equation (9). ▮

Intuitively, the first constraint in theorem 1 occurs because updates to a particular unknown must occur in convergence iteration order. Constraints 2 and 3 occur because within a convergence iteration, unknowns which share an edge in the matrix graph must be executed in the order provided by σ. Constraints 4 and 5 arise because for unknowns which share an edge in the matrix graph, the previous convergence iteration's update on a neighboring unknown must occur before the current convergence iteration of the current unknown.

3 Generating the Tiling and Reordering Functions via Sparse Tiling

Sparse tiling techniques subdivide the iteration space of a Gauss-Seidel computation (such as the one in Figure 2) to perform run-time data and iteration reordering. Recall from Section 1 that the Gauss-Seidel computation can be visualized as shown in Figure 2. The algorithms which generate a new data order and computation schedule at runtime manipulate the matrix graphs within the Gauss-Seidel computation. The output of the run-time component of sparse tiling (inspector phase) is a data reordering function, σ, a tiling function, θ, and a corresponding schedule function, *sched*.

Both sparse tiling techniques includes the following steps within the inspector phase at runtime.

- **Partition** the matrix graph to create a seed partitioning.
- **Grow tiles** from the cells of the seed partitioning to create a tiling function θ which assigns each iteration point to a tile. The tile growth algorithm also generates constraints on the data reordering function.
- **Generate** the reordering function σ.
- **Remap** the data using the reordering function σ.
- **Reschedule** by creating a schedule function, *sched*, based on the tiling function θ.

The sparse tiling inspector operates on the original matrix graph, $G(V, E)$. As described in Section 1, for each iteration v, there is a node in the matrix graph $v \in V$, and for each nonzero in the matrix, $a_{vw} \neq 0$ or $a_{wv} \neq 0$, there is a directed edge $<v, w> \in E$. When calculating the algorithmic complexity for the various algorithms which are part of the inspector, we assume that the matrix graph is stored in the CSR sparse matrix format. Therefore, accessing all edges $<v, *>$, with a particular node v as the first endpoint takes $O(|E|/|V|)$. Accessing all edges such that node v is the second endpoint, $<*, v>$ takes $O(|E|)$.

The next sub-sections describe each step of the run-time process for *full sparse tiling* (previously referred to as serial sparse tiling [21]) in terms of algorithmic complexity and provable characteristics of the resulting σ and θ functions. The only necessary difference between full sparse tiling and cache blocking [4] is the tile-growth algorithm. Characteristics of the resulting σ and θ functions are determined and then used in section 3.6 to show that full sparse tiled Gauss-Seidel generates results equivalent to typical Gauss-Seidel when the same data order σ is used by both.

3.1 Partition the Matrix Graph

Although optimal graph partitioning is an NP-Hard problem [6], there are many heuristics 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 Metis package [12] to generate the seed partitioning function $part$. The partitioning algorithm in Metis has a reported complexity of $O(|E|)$ where $|E|$ is the number of edges in the matrix graph. After the partitioning all the nodes in the matrix graph $v \in V$ have been assigned a seed partition, $part(v)$.

3.2 Sparse Tile the Iteration Space

The matrix graph partitioning, generated in the **Partition** step, creates a seed partitioning from which tiles can be grown. The seed partitioning designates the tiling at a particular convergence iteration, $iter_s$. In other words at $iter_s$, where $1 \leq iter_s \leq T$, the tiling function is set to the partition function, $\theta(iter_s, v) = part(v)$.

To determine the tiling at other convergence iterations full sparse tiling adds or deletes nodes from the seed partition to allow atomic execution of tiles across convergence iterations without violating any data dependences.

The FullSparseNaive_GSCSR Algorithm, shown in figure 7, generates the tiling function θ, which assigns iteration points to tiles. It also generates 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 $iter_s$. It then loops down through the convergence iterations that come before $iter_s$ setting the θ function for each iteration point $<iter, v>$ to the same as the iteration point directly above it in the iteration space. Finally, it visits the edges which have endpoints in two different partitions adjusting θ to ensure that the data dependences are satisfied by the θ values. The process is repeated for the convergence iterations between $iter_s$ and T in the upward tile growth. Once neighboring nodes are put into two different tiles at any iteration $iter$, 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 $iter$, $\theta(iter, v) < \theta(iter, w)$.

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 matrix graph, and $|E|$ is the number of edges in the matrix graph. The $k|V||E|$ term is due to the while loops at lines 5 and 15. In the worst case, the while loop will execute $k|V|$ times. $\forall v \in V$, $\theta(iter, v)$ decreases monotonically and can take on at most k values, and in the worst case only one $\theta(iter, v)$ would decrease each time through the while loop. In practice, the algorithm runs much faster than this bound.

Figures 8 and 9 list post-conditions for the FullSparseNaive_GSCSR algorithm. Next we give a proof for each of the post-conditions.

Post-condition 2.1 $\forall v \in V, \theta(iter_s, v)$ *is initialized*

Satisfied by assignments in line 1. ▮

Post-condition 2.2 $\forall v, w \in V$*,* $<v, w> \in NodeOrd$ *if and only if*
$\theta(iter_s, v) < \theta(iter_s, w)$ *and* $(<v, w> \in E$ *or* $<w, v> \in E)$

Satisfied by assignment in line 2. ▮

Post-condition 4.1 $\forall v \in V$*,* $\theta(iter, v) = \theta(iter + 1, v)$

```
Algorithm FullSparseNaive_GSCSR($G(V,E)$,$part()$,$T$,$iter_s$)

1:  foreach vertex $v \in V, \theta(iter_s, v) \leftarrow part(v)$
2:  $NodeOrd \leftarrow \{<v,w> \mid \theta(iter_s, v) < \theta(iter_s, w)$ and $(<v,w> \in E$ or $<w,v> \in E)\}$

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

Upward tile growth
13: for $iter = (iter_s + 1)$ to $T$
14:     foreach vertex $v \in V$, $\theta(iter, v) \leftarrow \theta(iter - 1, v)$
15:     do while $\theta$ changes
16:         foreach $<v,w> \in NodeOrd$
17:             $\theta(iter, v) \leftarrow max(\theta(iter, v), \theta(iter - 1, w))$
18:             $\theta(iter, w) \leftarrow max(\theta(iter, w), \theta(iter, v))$
19:         end foreach
20:     end do while
21:     $NodeOrd \leftarrow NodeOrd \bigcup$
                $\{<v,w> \mid \theta(iter, v) < \theta(iter, w)$ and $(<v,w> \in E$ or $<w,v> \in E)\}$
22: end for
```

Figure 7: FullSparseNaive_GSCSR Algorithm

Satisfied by assignment in line 4 and pre-condition 4.1. Pre-condition 4.1 is satisfied by post-condition 2.1 when $iter = (iter_s - 1)$ in the loop starting at line 3. For all $iter$ such that $1 \leq iter < (iter_s - 1)$, it is satisfied by the post-condition 4.1 from the previous iteration $(iter + 1)$. ∎

Post-condition 7.1 $\theta(iter, w) \leq \theta(iter + 1, w)$

Post-condition 4.1 ensures that $\theta(iter, w)$ is initialized equal to $\theta(iter + 1, w)$. In line 7, $\theta(iter, w)$ can only decrease in value thus post-condition 7.1 holds. ∎

Post-condition 7.2 $\theta(iter, w) \leq \theta(iter + 1, v)$

Satisfied by the assignment in line 7. ∎

```
Algorithm FullSparseNaive_GSCSR($G(V,E)$,$part()$,$T$,$iter_s$)

     { pre-condition [1.1] $(1 \leq iter_s \leq T)$}

 1:  foreach vertex $v \in V, \theta(iter_s, v) \leftarrow part(v)$
 2:  $NodeOrd \leftarrow \{<v,w> \mid \theta(iter_s, v) < \theta(iter_s, w)$ and $(<v,w> \in E$ or $<w,v> \in E)\}$

     { post-condition [2.1] $\forall v \in V, \theta(iter_s, v)$ is initialized }
     { post-condition [2.2] $\forall v, w \in V$, $<v,w> \in NodeOrd$ if and only if
                 $\theta(iter_s, v) < \theta(iter_s, w)$ and $(<v,w> \in E$ or $<w,v> \in E)$ }

Downward tile growth
 3:  for $iter = (iter_s - 1)$ downto 1

          { pre-condition [4.1] $\forall v \in V, \theta(iter + 1, v)$ is initialized }
 4:       foreach vertex $v \in V$, $\theta(iter, v) \leftarrow \theta(iter + 1, v)$
          { post-condition [4.1] $\forall v \in V$, $\theta(iter, v) = \theta(iter + 1, v)$}

 5:       do while $\theta$ changes
 6:            foreach $<v,w> \in NodeOrd$
 7:                 $\theta(iter, w) \leftarrow min(\theta(iter, w), \theta(iter + 1, v))$
                    { post-condition [7.1] $\theta(iter, w) \leq \theta(iter + 1, w)$}
                    { post-condition [7.2] $\theta(iter, w) \leq \theta(iter + 1, v)$}

 8:                 $\theta(iter, v) \leftarrow min(\theta(iter, v), \theta(iter, w))$
                    { post-condition [8.1] $\theta(iter, v) \leq \theta(iter + 1, v)$}
                    { post-condition [8.2] $\theta(iter, v) \leq \theta(iter, w)$}

 9:            end foreach
               { post-condition [9.1] $\forall v \in V$, $\theta(iter, v) \leq \theta(iter + 1, v)$}
               { post-condition [9.2] if $\theta$ didn't change then
                    $\forall <v,w> \in NodeOrd$, $\theta(iter, v) \leq \theta(iter, w) \leq \theta(iter + 1, v)$}

10:       end do while
          { post-condition [10.1] $\forall <v,w> \in NodeOrd$, $\theta(iter, v) \leq \theta(iter, w) \leq \theta(iter + 1, v)$}
          { post-condition [10.2] $\forall v, w \in V$, if $(<v,w> \in E$ or $<w,v> \in E)$
                    then $\theta(iter, v) \leq \theta(iter + 1, w)$}

11:       $NodeOrd \leftarrow NodeOrd \bigcup$
                 $\{<v,w> \mid \theta(iter, v) < \theta(iter, w)$ and $(<v,w> \in E$ or $<w,v> \in E)\}$

12:  end for
     { post-condition [12.1] $\forall iter : 1 \leq iter \leq (iter_s - 1)$ and $\forall v \in V$, $\theta(iter, v) \leq \theta(iter + 1, v)$}
     { post-condition [12.2] $\forall v, w \in V$, $<v,w> \in NodeOrd$ if and only if
                 $\exists iter : 1 \leq iter \leq (iter_s - 1)$ such that $\theta(iter, v) < \theta(iter, w)$
                 and $(<v,w> \in E$ or $<w,v> \in E)$ }
     { post-condition [12.3] $\forall iter : 1 \leq iter \leq (iter_s - 1)$ and $\forall v, w \in V$,
            if $(<v,w> \in E$ or $<w,v> \in E)$ then $\theta(iter, v) \leq \theta(iter + 1, w)$}
     { post-condition [12.4] $NodeOrd$ is acyclic }
```

Figure 8: FullSparseNaive_GSCSR Algorithm with post-conditions, Part I

```
Algorithm FULLSPARSENAIVE_GSCSR cont...


Upward tile growth
13: for $iter = (iter_s + 1)$ to $T$

        { pre-condition [14.1] $\forall v \in V$, $\theta(iter - 1, v)$ is initialized }
14:     foreach vertex $v \in V$, $\theta(iter, v) \leftarrow \theta(iter - 1, v)$
        { post-condition [14.1] $\forall v \in V$, $\theta(iter, v) = \theta(iter - 1, v)$}

15:     do while $\theta$ changes
16:         foreach $<v, w> \in NodeOrd$
17:             $\theta(iter, v) \leftarrow max(\theta(iter, v), \theta(iter - 1, w))$
                { post-condition [17.1] $\theta(iter - 1, v) \leq \theta(iter, v)$}
                { post-condition [17.2] $\theta(iter - 1, w) \leq \theta(iter, v)$}

18:             $\theta(iter, w) \leftarrow max(\theta(iter, w), \theta(iter, v))$
                { post-condition [18.1] $\theta(iter - 1, w) \leq \theta(iter, w)$}
                { post-condition [18.2] $\theta(iter, v) \leq \theta(iter, w)$}

19:         end foreach
            { post-condition [19.1] $\forall v \in V$, $\theta(iter - 1, v) \leq \theta(iter, v)$}
            { post-condition [19.2] if $\theta$ didn't change then
                $\forall <v, w> \in$ $NodeOrd$, $\theta(iter - 1, w) \leq \theta(iter, v) \leq \theta(iter, w)$}

20:     end do while
        { post-condition [20.1] $\forall <v, w> \in NodeOrd$, $\theta(iter - 1, w) \leq \theta(iter, v) \leq \theta(iter, w)$}
        { post-condition [20.2] $\forall v, w \in V$, if ($<v, w> \in E$ or $<w, v> \in E$)
                                              then $\theta(iter - 1, v) \leq \theta(iter, w)$}

21:     $NodeOrd \leftarrow NodeOrd \bigcup$
                      $\{<v, w> \mid \theta(iter, v) < \theta(iter, w)$ and ($<v, w> \in E$ or $<w, v> \in E$)$\}$

22: end for
    { post-condition [22.1] $\forall q : (iter_s + 1) \leq q \leq T$ and $\forall v \in V$, $\theta(q - 1, v) \leq \theta(q, v)$}
    { post-condition [22.2] $\forall q : (iter_s + 1) \leq q \leq T$ and $\forall v, w \in V$, if ($<v, w> \in E$ or $<w, v> \in E$)
                      then $\theta(q - 1, v) \leq \theta(q, w)$}
```

Figure 9: FULLSPARSENAIVE_GSCSR Algorithm with post-conditions, Part II

Post-condition 8.1 $\theta(iter, v) \leq \theta(iter + 1, v)$

Post-condition 4.1 ensures that $\theta(iter, v)$ is initialized equal to $\theta(iter + 1, v)$. In line 8, $\theta(iter, v)$ can only decrease in value thus post-condition 8.1 holds. ▌

Post-condition 8.2 $\theta(iter, v) \leq \theta(iter, w)$

Satisfied by the assignment in line 8. ▌

Post-condition 9.1 $\forall iter \in V$*,* $\theta(iter, v) \leq \theta(iter + 1, v)$

Satisfied by post-conditions 4.1, 7.1, and 8.1. ▌

Post-condition 9.2 *if* θ *didn't change then* $\forall <v, w> \in NodeOrd$*,* $\theta(iter, v) \leq \theta(iter, w) \leq \theta(iter + 1, v)$

Follows immediately from post-conditions 7.2 and 8.2. 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 $< v_1, v_2 > \in NodeOrd$ and $< v_2, v_3 > \in NodeOrd$ with $\theta(iter + 1, v_3) = 0, \theta(iter, v_1) = \theta(iter + 1, v_1) > 0$, and $\theta(iter, v_2) > 0$. If edge $< v_1, v_2 >$ is visited first in the foreach loop, then it will still be the case that $\theta(iter, v_1) > 0$ and $\theta(iter, v_2) > 0$. However, later in the foreach loop when $<v_2, v_3>$ is visited, $\theta(iter, v_2)$ will be set equal to 0 due to line 8. It will then be the case that $\theta(iter, v_1) > \theta(iter, v_2)$ even though $<v_1, v_2> \in NodeOrd$. This will be remedied the next time through the foreach loop. ▌

Post-condition 10.1 $\forall <v, w> \in NodeOrd$*,* $\theta(iter, v) \leq \theta(iter, w) \leq \theta(iter + 1, v)$

The do while loop in lines 5 through 10 ends when θ no longer changes in the foreach loop starting at line 6. Therefore due to post-condition 9.2, when the do while loop completes post-condition 10.1 is satisfied. ▌

Post-condition 10.2 $\forall v, w \in V$*, if* $(<v, w> \in E$ *or* $<w, v> \in E)$ *then* $\theta(iter, v) \leq \theta(iter + 1, w)$

For all $v, w \in V$ with $iter = (iter_s - 1)$, the tiling function values $\theta(iter + 1, v)$ and $\theta(iter + 1, w)$ are set in line 1. For all $v, w \in V$ with $iter$ such that $1 \leq iter < (iter_s - 1)$, the tiling function values $\theta(iter + 1, v)$ and $\theta(iter + 1, w)$ are set in the previous iteration of the for loop starting at line 3. The relationship between $\theta(iter + 1, v)$ and $\theta(iter + 1, w)$ falls under two cases, either $\theta(iter + 1, v) \leq \theta(iter + 1, w)$ or $\theta(iter + 1, v) > \theta(iter + 1, w)$.

Case 1: If $\theta(iter+1, v) \leq \theta(iter+1, w)$ then due to post-condition 9.1 the following is true and post-condition 10.2 is satisfied by the second inequality.

$$\theta(iter, v) \leq \theta(iter + 1, v) = \theta(iter + 1, w) \tag{10}$$

Case 2: If $\theta(iter + 1, v) > \theta(iter + 1, w)$ then due to post-condition 2.2 when $iter = (iter_s - 1)$ and line 11 when $1 \leq iter < (iter_s - 1)$, the following is true.

$$<w, v> \in NodeOrd \tag{11}$$

Using (11) and swapping the roles of v and w in post-condition 10.1, the following is true and post-condition 10.2 is satisfied.

$$\theta(iter, w) \leq \theta(iter, v) \leq \theta(iter + 1, w) \tag{12}$$

▮

Post-condition 12.1 *$\forall iter : 1 \leq iter \leq (iter_s - 1)$ and $\forall v \in V$, $\theta(iter, v) \leq \theta(iter + 1, v)$*

Satisfied by the loop bounds of the for loop starting at line 3 and post-condition 9.1. ▮

Post-condition 12.2 *$\forall v, w \in V$, $<v, w> \in NodeOrd$ if and only if*
$\exists iter : 1 \leq iter \leq iter_s$ such that $\theta(iter, v) < \theta(iter, w)$ and $(<v, w> \in E$ or $<w, v> \in E)$

Satisfied by post-condition 2.2 and the loop bounds of the for loop starting at line 3 combined with the assignment in line 11.

Post-condition 12.3 *$\forall iter : 1 \leq iter \leq (iter_s - 1)$ and $\forall v, w \in V$, if $(<v, w> \in E$ or $<w, v> \in E)$ then $\theta(iter, v) \leq \theta(iter + 1, w)$*

Satisfied by the loop bounds of the for loop starting at line 3 and post-condition 10.2. ▮

Post-condition 12.4 *$NodeOrd$ is acyclic*

Post-condition 2.2 quarantees that $NodeOrd$ is initialized as acyclic. During the downward tile growth, relations are added to $NodeOrd$ at line 11. We show that each new relation added at line 11 does not cause a cycle with the current set of relations in $NodeOrd$.

We assume the contrary and then derive a contradiction. Assume there is a path $< w, x_0 > ... < x_n, v >$ in $NodeOrd$ such that upon adding $<v, w>$ at line 11 a cycle would be created. Due to the first inequality in post-condition 10.1 and the assignment at line 11, the following statement is true about the tiling function θ values for the nodes in the path for the current value of $iter$ at line 11.

$$\theta(iter, w) \leq \theta(iter, x_0) \leq \cdots \leq \theta(iter, x_n) \leq \theta(iter, v) \tag{13}$$

Due to line 11, if the relation $<v, w>$ is being added to $NodeOrd$ then the following is true.

$$\theta(iter, v) < \theta(iter, w) \tag{14}$$

Combining (13) and (14) results in the contradiction that $\theta(iter, v) < \theta(iter, v)$. Therefore, it is not possible to add a relation $<v, w>$ to $NodeOrd$ which will cause a cycle. ▮

Post-condition 14.1 *$\forall v \in V$, $\theta(iter, v) = \theta(iter - 1, v)$*

Satisfied by assignment in line 14 and pre-condition 14.1. Pre-condition 14.1 is satisfied by post-condition 2.1 when $iter = (iter_s + 1)$ in the loop starting at line 13. For all $iter$ such that $(iter_s + 1) > iter \leq T$, it is satisfied by the post-condition 14.1 from the previous iteration $(iter - 1)$. ▮

Post-condition 17.1 $\theta(iter - 1, v) \leq \theta(iter, v)$

Post-condition 14.1 ensures that $\theta(iter, v)$ is iniatilized equal to $\theta(iter - 1, v)$. In line 17, $\theta(iter, v)$ can only increase in value thus postcondition 17.1 holds. ▌

Post-condition 17.2 $\theta(iter - 1, w) \leq \theta(iter, v)$

Satisfied by the assignment in line 17. ▌

Post-condition 18.1 $\theta(iter - 1, w) \leq \theta(iter, w)$

Post-condition 14.1 ensures that $\theta(iter, w)$ is initialized equal to $\theta(iter - 1, w)$. In line 18, $\theta(iter, w)$ can only increase in value thus postcondition 18.1 holds. ▌

Post-condition 18.2 $\theta(iter, v) \leq \theta(iter, w)$

Satisfied by the assignment in line 18. ▌

Post-condition 19.1 $\forall v \in V,$ $\theta(iter - 1, v) \leq \theta(iter, v)$

Satisfied by post-conditions 14.1, 17.1, and 18.1. ▌

Post-condition 19.2 *if* θ *didn't change then* $\forall <v, w> \in NodeOrd,$ $\theta(iter - 1, w) \leq \theta(iter, v) \leq \theta(iter, v)$

Follows immediately from post-conditions 17.2 and 18.2. 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 $<w_1, w_2> \in NodeOrd$ and $<w_2, w_3> \in NodeOrd$ with $\theta(iter - 1, w_1) = \theta(iter, w_1) = 4, \theta(iter - 1, w_2) = \theta(iter, w_2) < 4$, and $\theta(iter, w_3) < 4$. If edge $<w_2, w_3>$ is visited first in the foreach loop, then after lines 17 and 18 it will still be the case that $\theta(iter, w_2) < 4$ and $\theta(iter, w_3) < 4$. However, later in the foreach loop when $<w_1, w_2>$ is visited, $\theta(iter, w_2)$ will be set equal to 4 due to line 18. It will then be the case that $\theta(iter, w_2) > \theta(iter, w_3)$ even though $<w_3, w_3> \in NodeOrd$. This will be remedied the next time through the foreach loop. ▌

Post-condition 20.1 $\forall <v, w> \in NodeOrd,$ $\theta(iter - 1, w) \leq \theta(iter, v) \leq \theta(iter, w)$

The do while loop in lines 15 through 20 ends when θ no longer changes in the foreach loop starting at line 18. Therefore due to post-condition 19.2, when the do while loop completes post-condition 20.1 is satisfied. ▌

Post-condition 20.2 $\forall v, w \in V,$ *if* $(<v, w> \in E$ *or* $<w, v> \in E)$ *then* $\theta(iter - 1, v) \leq \theta(iter, w)\}$

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

Case 1: If $\theta(iter - 1, v) \leq \theta(iter - 1, w)$ then due to post-condition 19.1 the following is true and post-condition 20.2 is satisfied.

$$\theta(iter - 1, v) \leq \theta(iter - 1, w) \leq \theta(iter, v) \tag{15}$$

Case 2: If $\theta(iter - 1, v) > \theta(iter - 1, w)$ then due to post-conditions 12.2 when $iter = (iter_s + 1)$ and line 11 if $(iter_s + 1) < iter \leq T$, the following is true.

$$< w, v > \in NodeOrd \tag{16}$$

Using (16) and swapping the roles of v and w in post-condition 20.1, we find the following is true and post-condition 20.2 is satisfied by the second inequality.

$$\theta(iter - 1, v) \leq \theta(iter, w) \leq \theta(iter, v) \tag{17}$$

▌

Post-condition 22.1 $\forall q : (iter_s + 1) \leq q \leq T$ *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 13 and post-condition 19.1. ▌

Post-condition 22.2 $\forall q : (iter_s + 1) \leq q \leq T$ *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 13 and post-condition 20.2. ▌

The post-conditions for the FullSparseNaive_GSCSR algorithm allow for the following lemmas, which describe the conditions met by tiling function θ and the set of ordered pairs $NodeOrd$ at the end of the algorithm.

Lemma 1 *Upon completion of the* FullSparseNaive_GSCSR *algorithm,* $\forall iter : 1 \leq iter \leq (T - 1)$ *and* $\forall v \in V$, $\theta(iter, v) \leq \theta(iter + 1, v)$.

This condition states that all later convergence iterations performed on the same node v will be in the same or later tile. It depends directly on post-conditions 12.1, and 22.1, which are post-conditions for the downward tile growth and upward tile growth sections of the algorithm respectively. Between lines 12 and the end of the algorithm, no assignments occur to $\theta(iter, v)$ with $1 \leq iter \leq iter_s$ and $v \in V$.

Notice that upon substitution of $q = iter + 1$ in post-condition 22.1 we get the following statement.

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

Rewriting (18) we get the following.

$$\forall iter : iter_s \leq iter \leq (T - 1) \quad \text{and} \quad \forall v \in V, \quad \theta(iter, v) \leq \theta(iter + 1, v) \tag{19}$$

By combining the domains of *iter* in post-condition 12.1 and (19) we get post-condition 1. ▮

Lemma 2 *Upon completion of the* FULLSPARSENAIVE_GSCSR *algorithm,* $\forall v, w \in V$, $<v, w> \in NodeOrd$ *if and only if* $\exists iter : 1 \leq iter \leq T$ *such that* $\theta(iter, v) < \theta(iter, w)$ *and* ($<v, w> \in E$ *or* $<w, v> \in E$).

This condition states that for every ordered pair $<v, w>$ in the relation $NodeOrd$, the tiling function for these nodes at one or more of the convergence iterations satisfies a less than relationship and vice versa. Satisfied by post-condition 12.2 and the loop bounds of the for loop starting at line 13 combined with the assignment in line 21. ▮

Lemma 3 *Upon completion of the* FULLSPARSENAIVE_GSCSR *algorithm,* $\forall iter : 1 \leq iter \leq (T-1)$ *and* $\forall v, w \in V$, *if* ($<v, w> \in E$ *or* $<w, v> \in E$) *then* $\theta(iter, v) \leq \theta(iter+1, w)$.

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 12.3 and 22.2 which are post-conditions for the downward tile growth and upward tile growth sections of the algorithm respectively. Between line 12 and the end of the algorithm, no assignments occur to $\theta(iter, v)$ with $1 \leq iter \leq iter_s$ and $v \in V$.

Notice that upon substitution of $q = iter + 1$ in post-condition 22.2 we get the following statement.

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

Rewriting (20) we get the following.

$$\forall iter : iter_s \leq iter \leq (T-1) \quad \text{and} \quad \forall <v, w> \in E, \quad \theta(iter, v) \leq \theta(iter+1, w) \tag{21}$$

By combining the domains of *iter* in post-condition 12.3 and (21) post-condition 3 is satisfied. ▮

Lemma 4 *Upon completion of the* FULLSPARSENAIVE_GSCSR *algorithm,* $NodeOrd$ *is acyclic.*

Post-condition 12.4 guarantees that $NodeOrd$ is acyclic before the beginning of the for loop which starts at line 13. During the upwards tile growth, relations are added to $NodeOrd$ at line 21. We show that each new relation added at line 21 does not cause a cycle with the current set of relations in $NodeOrd$.

We assume the contrary and then derive a contradiction. Assume there is a path $< w, x_0 > ... < x_n, v >$ in the $NodeOrd$ such that upon adding $< v, w >$ at line 21 a cycle would be created. Due to the second inequality in post-condition 20.1 and the assignment at line 21, the following statement is true about the tiling function θ values for the nodes in the path for the current value of *iter* at line 21.

$$\theta(iter, w) \leq \theta(iter, x_0) \leq \cdots \leq \theta(iter, x_n) \leq \theta(iter, v) \tag{22}$$

Due to line 21, if the relation $<v, w>$ is being added to $NodeOrd$ then the following is true.

$$\theta(iter, v) < \theta(iter, w) \tag{23}$$

Combining (22) and (23) result in the contradiction that $\theta(iter, v) < \theta(iter, v)$. Therefore, it is not possible to add a relation $<v, w>$ to $NodeOrd$ which will cause a cycle. ▮

3.3 Generate the Reordering Function

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 unknown computation will use the most recent values of the neighboring nodes in the matrix graph. Within a convergence iteration unknowns are updated based on their order.

We have two goals when ordering the unknowns: satisfy the constraints specified in the $NodeOrd$ relation and increase intra-iteration locality. First and foremost, the reordering function must satisfy the $NodeOrd$ relation for correctness. Second, we want to give consecutive numbers to unknowns that at any iteration are updated by the same tile, because the data 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 as well as inter-iteration locality.

Both of the above goals are satisfied when the nodes in the matrix graph (and associated unknowns) are ordered based on the lexicographical order of their tile vectors. The *tile vector* is a vector of length T which indicates the tile function values for a given node v at each convergence iteration, $< \theta(1, v), ...\theta(T, v) >$. Our current implementation uses quicksort to sort the nodes lexicographically according to their tile vector. Since each comparison between tile vectors requires $O(T)$ time, the complexity of quicksort in this instance is $O(T|V|lg|V|)$. A radix sort with complexity $O(T(|k| + |V|))$ is also possible. Creating the reordering function σ by sorting the nodes based on their tile vectors results in the following property.

Lemma 5 *After constructing an ordering on the nodes in the matrix graph based on the lexicographical ordering of their tile vectors, if* $<v, w> \in NodeOrd$ *then* $\sigma(v) < \sigma(w)$

Proof:

If $< v, w > \in NodeOrd$ then due to Lemma 2 $\exists iter$ such that $\theta(iter, v) < \theta(iter, w)$ and $\not\exists iter$ such that $\theta(iter, w) < \theta(iter, v)$. Therefore, the tile vector $< \theta(1, v), \cdots, \theta(T, v) >$ lexicographically precedes the tile vector $< \theta(1, w), \cdots, \theta(T, w) >$. Since σ is created with a lexicographical sort of the tile vectors, it follows that $\sigma(v) < \sigma(w)$. ▌

3.4 Reorder the Data

The σ function generated by the CreateSigma algorithm is used to remap the data from the original unknown array $\vec{u}$ to a new vector $\vec{u'}$ such that $u'_{\sigma(v)} = u_v$. Also the the rows and columns of the sparse matrix are remapped such that $A'_{\sigma(v)\sigma(w)} = A_{vw}$. The complexity of the data remap is $O(|V| + |E|)$.

3.5 Create Schedule

To generate the schedule function *sched* such that $sched(tileID, iter) = \{\sigma(v) \mid \theta(iter, v) = tileID\}$, it is necessary to traverse all the iteration points in Gauss-Seidel without actually doing the computation. The complexity for generating the schedule function is $O(T|V|)$.

3.6 Proof of Correctness

By using Lemmas 1 through 5, which describe the characteristics of the tiling function θ and the reordering function σ generated by the full sparse tiling algorithms, we are able to prove that sparse tiled Gauss-Seidel in Figure 6 generate bit-equivalent results to those generated by Gauss-Seidel for CSR in Figure 5 when both use σ for their initial data ordering.

Theorem 2 *Let $G(V, E)$ be the directed matrix graph for a square sparse matrix A. For each row v in the matrix, there is a corresponding node in the graph, $v \in V$. For each element in the matrix, A_{vw}, there is an edge $< v, w > \in E$. The reordered unknowns $\vec{u'}$, right-hand side $\vec{f'}$ and sparse matrix A', are such that $i = \sigma(v)$, $A'_{\sigma(v)\sigma(w)} = A_{vw}$, $u'_{\sigma(v)} = u_v$, and $f'_{\sigma(v)} = f_v$. A' is represented with the arrays* **ia**, **ja**, *and* **a** *such that for each $A'_{\sigma(v)\sigma(w)}$ there exists p such that* $\mathbf{ia}[\sigma(v)] \leq p < \mathbf{ia}[\sigma(v) + 1]$ *and* $\sigma(w) = \mathbf{ja}[p]$.

The θ and σ functions, generated by the full sparse tiling inspector using FullSparseNaive_GSCSR *as the tile growth algorithm, satisfy the constraints in Theorem 1.*

Proof:

First, when using the facts that $i = \sigma(v)$ and for each $A'_{\sigma(v)\sigma(w)}$ there exists an edge $< v, w > \in E$ in the original sparse matrix graph for A, the constraints from Theorem 1 can be rewritten as follows:

1. $\forall\, iter_1, iter_2, v : (1 \leq iter_1 < iter_2 \leq T) \wedge (0 \leq \sigma(v) < R) \Rightarrow \theta(iter_1, v) \leq \theta(iter_2, v)$

2. $\forall\, iter, v, w : (1 \leq iter \leq T) \wedge (0 \leq \sigma(v) < \sigma(w) < R) \wedge < v, w > \in E \Rightarrow \theta(iter, v) \leq \theta(iter, w)$

3. $\forall\, iter, v, w :\ (1 \leq iter \leq T) \wedge (0 \leq \sigma(v) < \sigma(w) < R) \wedge < w, v > \in E \Rightarrow \theta(iter_1, v) \leq \theta(iter, w)$

4. $\forall\, iter_1, iter_2, v, w :\ (1 \leq iter_1 < iter_2 \leq T) \wedge (0 \leq \sigma(v) \neq \sigma(w) < R) \wedge < v, w > \in E \Rightarrow \theta(iter_1, v) \leq \theta(iter_2, w)$

5. $\forall\, iter_1, iter_2, v, w :\ (1 \leq iter_1 < iter_2 \leq T) \wedge (0 \leq \sigma(v) \neq \sigma(w) < R) \wedge < w, v > \in E \Rightarrow \theta(iter_1, v) \leq \theta(iter_2, w)$

Condition 1: Satisfied by transitive closure on Lemma 1.

Condition 2: For condition 2, we assume the contrary and derive a contradiction. Assume the following:

$$\forall\, iter, v, w : (1 \leq iter \leq T) \wedge (0 \leq \sigma(v) < \sigma(w) < R) \wedge < v, w > \in E \wedge \theta(iter, v) > \theta(iter, w) \qquad (24)$$

Using Lemma 2 and the assumption from (24) that $\theta(iter, v) > \theta(iter, w)$, it is true that $< w, v > \in NodeOrd$. Therefore according to Lemma 5, $\sigma(w) < \sigma(v)$ which contradicts (24).

Condition 3: Can be shown with the same proof that was used for Condition 2.

Condition 4: Satisfied by transitive closure on Lemma 3.

Condition 5: Satisfied by transitive closure on Lemma 3. ▌

4 Full Sparse Tiling Efficiency Issues

The **Partition**, **Grow Tiles**, **Generate Sigma**, **Reorder**, and **Reschedule** steps account for the runtime overhead of sparse tiling. Since all of these steps occur at runtime, their efficiency is important. The tile growth algorithm FullSparseNaive_GSCSR has complexity $O(Tk|V||E|)$, where T is the number of convergence iterations, k is the number of tiles, $|V|$ is the number of nodes in the matrix graph, and $|E|$ is the number of edges in the matrix graph. Figure 10 shows the tile growth algorithm FullSparseWorkSet_GSCSR with complexity $O(T^2k|E|)$. Since T is typically much less than $|V|$ this algorithm has better worst-case complexity.

Consider only the downward tile growth phase (the argument for the upward tile growth is similar). In the FullSparseNaive_GSCSR algorithm the while loop at line 5 is necessary because a specific $\theta(iter, v)$ could change multiple times. Such a change occurs if a relation $< v, w >$ is in $NodeOrd$ and $\theta(iter, w)$ changes due to a relation $<w, z>\in NodeOrd$ which is visited later than $<v, w>$ in the foreach loop. The FullSparseWorkSet_GSCSR algorithm avoids the need for the while loop by incorporating two changes. First FullSparseWorkSet_GSCSR has two loops, at lines 7 and 11, over the relations in $NodeOrd$. The first loop makes sure that if node v comes before node w in the $NodeOrd$ relation, that the iteration point $< iter, w >$ must be in the same or an earlier tile than the iteration point $< iter + 1, v >$. Since the tiling function values for all nodes $v \in V$ won't change at the $(iter + 1)$ iteration, it is only necessary to visit each $< v, w >\in NodeOrd$ once to get $\theta(iter, w) \leq \theta(iter + 1, v)$. The second loop through the relations in $NodeOrd$ makes sure that if $< v, w >\in NodeOrd$ then iteration point $< iter, v >$ is put into the same or earlier tile as iteration point $< iter, w >$. Since we visit the $NodeOrd$ relations $< v, w >$ in order of the current tiling function value for w, $\theta(iter, w)$, at any node v there will not be a path in $NodeOrd$, $< v, w >, < w, x_1 >, ..., < x_{n-1}, x_n >$ such that $\theta(iter, x_n) < \theta(iter, w)$. Therefore, it is only necessary to visit each $< v, w >\in NodeOrd$ once in the second loop as well. Upon elimination of the while loop, the complexity of the algorithm changes from $O(Tk|V||E|)$ to $O(T^2k|E|)$, where the extra T term is due to the UpdateTheta algorithm and the k term is due to the loop which starts at line 11.

Another costly part of the FullSparseNaive_GSCSR algorithm occurs at lines 11 and 21, where each edge $<v, w>$ in the matrix graph must be checked to determine if $<v, w>$ belongs in $NodeOrd$. If $<v, w>$ is not in $NodeOrd$ it must be the case that $\theta(iter+1, v) \geq \theta(iter+1, w)$. Thus, it is only necessary to check an edge $< v, w >$ if either $\theta(iter, v)$ has decreased during downward tile growth or $\theta(iter, w)$ has increased during forward tile growth. In FullSparseWorkSet_GSCSR, $ThetaChangedWorkSet$ keeps track of all nodes v whose $\theta(iter, v)$ value has changed. The nodes in the $ThetaChangedWorkSet$ are then used to determine which edges $<v, w>\in E$ must be checked for candidacy in $NodeOrd$. Since the upper bound on the number of edges checked in lines 22 and 42 is $|E|$, the worst-case complexity doesn't improve due to this change.

To show that the FullSparseWorkSet_GSCSR algorithm also satisfies the constraints specified in Theorem 1, it is only necessary to show that FullSparseWorkSet_GSCSR satisfies the same constraints as FullSparseNaive_GSCSR which were where shown in Lemmas 1-4.

Lemma 6 *Upon completion of the* FullSparseWorkSet_GSCSR *algorithm,* $\forall iter : 1 \leq iter \leq (T - 1)$ *and* $\forall v \in V$*,* $\theta(iter, v) \leq \theta(iter + 1, v)$*.*

All the tiling function values θ for a particular node are initialized in line 1 of the FullSparseWorkSet_GSCSR algorithm to the same value. During backward tile growth, the θ values for a node only change at lines 9 and 17 and can only decrease. During forward tile growth, the θ values for a node only change at lines 29 and 37 and can only increase. ▌

Lemma 7 *Upon completion of the* FullSparseWorkSet_GSCSR *algorithm,* $\forall v, w \in V$*,* $<v,w>\in NodeOrd$ *if and only if* $\exists iter : 1 \le iter \le T$ *such that* $\theta(iter, v) < \theta(iter, w)$ *and* $(<v,w> \in E$ *or* $<w,v> \in E)$*.*

The statement $\forall v, w \in V$, if $<v,w>\in NodeOrd$ then $\exists iter : 1 \le iter \le T$ such that $\theta(iter, v) < \theta(iter, w)$, is satisfied by the constraints on edges added to the $NodeOrd$ set in lines 3, 22 and 42.

The other direction of the equivalence is as follows.

$$\forall v, w \in V, \text{ if } \exists iter : 1 \le iter \le T \text{ such that } \theta(iter, v) < \theta(iter, w) \text{ and } (<v,w> \in E \text{ or } <w,v> \in E) \text{ then } <v,w>\in NodeOrd$$

First we show that after each iteration of backward tile growth if $\theta(iter, v) < \theta(iter, w)$ and $<v,w> \in E_{sym}$ then $<v,w>$ is put into the $NodeOrd$ set.

$$\forall <v,w> \in E_{sym}, 1 \le iter \le iter_s \text{ if } \theta(iter, v) < \theta(iter, w) \text{ then } <v,w>\in NodeOrd \tag{25}$$

Proposition (25) is true for $iter = iter_s$ due to line 3 in the algorithm. Assume the following is true.

$$\forall <x,y> \in E_{sym}, 1 \le iter < iter_s, \text{ if } \theta(iter+1, x) < \theta(iter+1, y) \text{ then } <x,y>\in NodeOrd \tag{26}$$

The following is equivalent to statement (26).

$$\forall <x,y> \in E_{sym}, 1 \le iter < iter_s, \text{ if } <x,y>\notin NodeOrd \text{ then } \theta(iter+1, x) \ge \theta(iter+1, y) \tag{27}$$

Due to line 1 in the algorithm and the calls to UpdateTheta, at the start of the loop in lines 5-23 if $\theta(iter+1, x) \ge \theta(iter+1, y)$ then $\theta(iter, x) \ge \theta(iter, y)$. The only way for the inequality to change to $\theta(iter, x) < \theta(iter, y)$ would be if $\theta(iter, x)$ is reduced at lines 9 or 17. If that occurs then x is put in the set $ThetaChangedWorkSet$. Due to line 22, if $x \in ThetaChangedWorkSet$ and $\theta(iter, x) < \theta(iter, y)$ and $<x,y>\in E_{sym}$, then $<x,y>$ is put into the relation $NodeOrd$. Therefore (25) is true for $iter$ such that $1 \le iter < iter_s$.

A similar argument may be made for forward tile growth using the following assumption.

$$\forall <x,y> \in E_{sym}, iter_s < iter \le T, \text{ if } \theta(iter-1, x) < \theta(iter-1, y) \text{ then } <x,y>\in NodeOrd \tag{28}$$

▌

Lemma 8 *Upon completion of the* FullSparseWorkSet_GSCSR *algorithm,* $\forall iter : 1 \le iter \le (T-1)$ *and* $\forall v, w \in V$*, if* $(<v,w> \in E$ *or* $<w,v> \in E)$ *then* $\theta(iter, v) \le \theta(iter+1, w)$*.*

The proof is similar to the one given in Lemma 3. It depends on the following post-conditions after line 16 and line 35 in the FullSparseWorkSet_GSCSR algorithm.

$$\forall v, w \in V, \text{ if } <v,w> \in E_{sym} \text{ then } \theta(iter, v) \leq \theta(iter+1, w)\} \tag{29}$$

$$\forall v, w \in V, \text{ if } <v,w> \in E_{sym} \text{ then } \theta(iter-1, v) \leq \theta(iter, w) \tag{30}$$

Proof for (29):

This proof is similar in form to that of Post-condition 10.2 from the FullSparseNaive_GSCSR algorithm. If $\theta(iter+1, v) \leq \theta(iter+1, w)$, due to lemma 6, $\theta(iter) < \theta(iter+1, w)$. For the case where $\theta(iter+1, v) > \theta(iter+1, w)$ due to lemma 7, it is true that $<w, v> \in NodeOrd$. Switching places with v and w at line 7, it will be the case that in line 9 there will be an update such that $\theta(iter, v) < \theta(iter+1, w)$.

Proof for (30):

This proof is similar in form to that of Post-condition 20.2 from the FullSparseNaive_GSCSR algorithm. If $\theta(iter-1, v) \leq \theta(iter-1, w)$ then due to lemma 6, $\theta(iter-1, v) \leq \theta(iter, w)$. For the case where $\theta(iter-1, v) > \theta(iter-1, w)$ due to lemma 7, it is true that $<w, v> \in NodeOrd$. Switching places with v and w at line 27, it will be the case that in line 29 there will be an update such that $\theta(iter-1, v) \leq \theta(iter, w)$.

▌

Lemma 9 *Upon completion of the* FullSparseWorkSet_GSCSR *algorithm, $NodeOrd$ is acyclic.*

The proof is similar to that of Post-condition 12.4 and Lemma 4. Essentially $NodeOrd$ is initialized as acyclic, and at lines 22 and 42 in the FullSparseWorkSet_GSCSR algorithm relation $<v, w>$ is only added to $NodeOrd$ when $\theta(iter, v) < \theta(iter, w)$ and $<v, w> \in E_{sym}$. The loops starting at lines 7, 11, 27, and 31 make sure that if $\exists iter_1$ such that $\theta(iter_1, v) < \theta(iter_1, w)$, then $\forall iter_2 \neq iter_1 : \theta(iter_2, v) \leq \theta(iter_2, w)$. ▌

```
Algorithm FullSparseWorkSet_GSCSR($G(V,E)$,$part()$,$T$,$iter_s$)

$\forall v \in V$ and $1 \leq iter \leq T, \theta(iter, v) \leftarrow part(v)$
$G_{sym}(V_{sym}, E_{sym}) = MakeSymmetricGraph(G(V,E))$
$NodeOrd \leftarrow \{<v,w> \mid \theta(iter_s, v) < \theta(iter_s, w)$ and $<v,w> \in E_{sym}\}$

Downwards tile growth
$\forall\ 0 \leq t < numtile$, $TileWorkSet_t \leftarrow \emptyset$
for $iter = iter_s - 1$ downto 1
    $ThetaChangedWorkSet \leftarrow \emptyset$
    foreach $<v,w> \in NodeOrd$
        $TileWorkSet_{\theta(iter,w)} \leftarrow TileWorkSet_{\theta(iter,w)} \cup \{w\}$
        if $\theta(iter,w) > \theta(iter+1,v)$ then UpdateTheta($1, iter, w, \theta(iter+1,v)$)
    end foreach
    for $t = 0$ to $(numtile - 1)$
        $toCheck \leftarrow TileWorkSet_t$
        while ($toCheck \neq \emptyset$)
            $tempSet \leftarrow toCheck$; $toCheck \leftarrow \emptyset$
            foreach $w \in tempSet$; foreach $<v,w> \in NodeOrd$
                if $\theta(iter,v) > \theta(iter,w)$ then
                    UpdateTheta($1, iter, v, \theta(iter,w)$)
                    $toCheck \leftarrow toCheck \cup \{v\}$
            end foreach; end foreach
        end while
    end for
    $NodeOrd \leftarrow NodeOrd \cup \{<v,w> \mid \theta(iter,v) < \theta(iter,w)$
                                and $v \in ThetaChangedWorkSet$ and $<v,w> \in E_{sym}\ \}$
end for

Upwards tile growth
$\forall\ 0 \leq t < numtile$, $TileWorkSet_t \leftarrow \emptyset$
for $iter = iter_s + 1$ to $T$
    $ThetaChangedWorkSet \leftarrow \emptyset$
    foreach $<v,w> \in NodeOrd$
        $TileWorkSet_{\theta(iter,v)} \leftarrow TileWorkSet_{\theta(iter,v)} \cup \{v\}$
        if $\theta(iter,v) < \theta(iter-1,w)$ then UpdateTheta($iter, T, v, \theta(iter-1,w)$)
    end foreach
    for $t = (numtile - 1)$ downto 0
        $toCheck \leftarrow TileWorkSet_t$
        while ($toCheck \neq \emptyset$)
            $tempSet \leftarrow toCheck$; $toCheck \leftarrow \emptyset$
            foreach $w \in tempSet$; foreach $<v,w> \in NodeOrd$
                if $\theta(iter,w) < \theta(iter,v)$ then
                    UpdateTheta($iter, T, w, \theta(iter,v)$)
                    $toCheck \leftarrow toCheck \cup \{w\}$
            end foreach; end foreach
        end while
    end for
    $NodeOrd \leftarrow NodeOrd \cup \{<v,w> \mid \theta(iter,v) < \theta(iter,w)$
                                and $w \in ThetaChangedWorkSet$ and $<v,w> \in E_{sym}\ \}$
end for
```

Figure 10: FullSparseWorkSet_GSCSR Algorithm

```
Algorithm MakeSymmetricGraph($G(V, E)$)

$V_{sym} = \emptyset$ and $E_{sym} = \emptyset$
$\forall v \in V,\ V_{sym} = V_{sym} \cup \{v\}$
foreach $<v, w> \in E,\ E_{sym} = E_{sym} \cup \{<v, w>\} \cup \{<w, v>\}$
return $G_{sym}(V_{sym}, E_{sym})$
```

Figure 11: MakeSymmetricGraph Algorithm

```
Algorithm UpdateTheta($start$,$end$,$x$,$newval$)

$oldval \leftarrow \theta(start, x)$
$TileWorkSet_{oldval} \leftarrow TileWorkSet_{oldval} - \{x\}$
$TileWorkSet_{newval} \leftarrow TileWorkSet_{newval} \cup \{x\}$
$ThetaChangedWorkSet \leftarrow ThetaChangedWorkSet \cup \{x\}$
$\forall\ start \leq q \leq end, \theta(q, x) \leftarrow newval$
```

Figure 12: UpdateTheta Algorithm

5 Related Work

Related work may 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. Another important distinction is between code transformations which have been automated in a compiler and those which are programming techniques which require some domain-specific knowledge.

Traditional tiling [25, 11, 5, 24, 23, 2, 15] may be applied to any perfect loop nest which traverses the unknowns associated with a regular mesh. This is because with regular grids the sparse matrix nonzeros are stored in 2D or 3D arrays and therefore the memory references and loop boundaries are affine. With the enabling transformation skewing, tiling is applicable to Gauss-Seidel and SOR over a regular mesh. when the nonzeros in the sparse matrix are stored 2D and 3D arrays. Although the code may be transformed to execute iteration points in a tile-by-tile fashion, the unknowns cannot be reordered for the Gauss-Seidel and SOR methods with a compile-time technique because currently there is no mechanism for identifying the domain specific information involved in reordering the unknown vector.

For Jacobi over a regular mesh, tiling techniques developed for imperfectly nested loops may be used [20, 1]. Another issue involved in tiling computations on regular meshes is how to determine the tiling and array padding parameters [18]. If other compiler transformations, such as skewing, function in-lining and converting while loops to for loops, are used then it is possible to apply tiling techniques for imperfectly nested loops to stationary iterative methods to achieve inter-iteration locality.

There has also been work on compiler generated inspectors/executors for improving intra-iteration locality [17, 3, 8, 16]. These papers describe how a compiler may analyze non-affine array references in a loop and generate the inspectors/executors for performing run-time data and iteration reordering. These techniques can be applied to the inner loops of Jacobi implemented for sparse matrix formats, but not to Gauss-Seidel or SOR due to the intra-iteration data dependences.

Im and Yelick [9, 10] use a code generator called SPARSITY to create sparse matrix-vector multiply code with better spatial and temporal locality when addressing the arrays for the $\vec{x}$ and $\vec{b}$ vectors in the system $A\vec{x} = \vec{b}$.

Increasing inter-iteration locality through programming techniques for iterative computations on regular meshes is explored by [4], [19], and [7].

The only other technique to our knowledge which handles inter-iteration locality for irregular meshes is unstructured cache blocking by Douglas et al. [4]. Cache blocking and full sparse tiling are the programming techniques which this paper refers to as sparse tiling techniques. The cache blocking results in [4] were generated within the context of a Multigrid algorithm using Gauss-Seidel. Within this context the authors are able to assume that the sparse matrices involved are symmetric and multiple rows in the sparse matrix will derive their structure from one node in an irregular mesh. Our work compares full sparse tiling and cache blocking in terms of sparse tiled Gauss-Seidel performance. Also, we look at overhead without the assumption of sparse matrix symmetry and structure derived from a mesh. Although these techniques differ in how the tiles are grown, the serial performance of Gauss-Seidel after applying either of these techniques is similar. Finally, the axiomatic approach we use to prove the correctness of the full sparse tiling tile growth algorithm could also be used to prove the correctness of the cache blocking tile growth algorithm.

6 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 data ordering. Finally, we present another version of the full sparse tile growth algorithm which has reduced worst-case complexity.

7 Acknowledgements

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

References

[1] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Conference Proceedings of the 2000 International Conference on Supercomputing*, pages 141–152, Santa Fe, New Mexico, May 8–11, 2000. ACM SIGARCH.

[2] Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of 1992 Conference on Supercomputing*, pages 114–124. IEEE Computer Society Press, November 1992.

[3] 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 1999 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices, pages 229–241, May 1–4, 1999.

[4] 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*, 10:21–40, February 2000.

[5] 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.

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

[7] Kang Su Gatlin and Larry Carter. Architecture-cognizant divide and conquer algorithms. In *Proceedings of 1999 Conference on Supercomputing*. ACM/IEEE, November 1999.

[8] 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.

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

[10] Eun-Jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In V.N.Alexandrov, J.J. Dongarra, B.A.Juliano, R.S.Renner, and C.J.K.Tan, editors, *Computational Science - ICCS 2001*, Lecture Notes in Computer Science. Springer, May 28-30, 2001.

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

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

[13] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, pages 107–124. Springer, August 8–10, 1994.

[14] Wayne Kelly and William Pugh. A unifying framework for iteration reordering transformations. Technical Report CS-TR-3430, University of Maryland, College Park, February 1995.

[15] 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.

[16] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, June 2001.

[17] 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*, pages 192–202, October 12–16, 1999.

[18] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of 2000 Conference on Supercomputing*, pages 60–61. ACM/IEEE, November 2000.

[19] Sriram Sellappa and Siddhartha Chatterjee. Cache-efficient multigrid algorithms. In V.N.Alexandrov, J.J. Dongarra, B.A.Juliano, R.S.Renner, and C.J.K.Tan, editors, *Computational Science - ICCS 2001*, Lecture Notes in Computer Science. Springer, May 28-30, 2001.

[20] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 215–228, May 1–4, 1999.

[21] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Rescheduling for locality in sparse matrix computations. In V.N.Alexandrov, J.J. Dongarra, B.A.Juliano, R.S.Renner, and C.J.K.Tan, editors, *Computational Science - ICCS 2001*, Lecture Notes in Computer Science. Springer, May 28-30, 2001.

[22] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, July 2002.

[23] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, volume 26 of *ACM SIGPLAN Notices*, pages 30–44, June 1991.

[24] M. Wolfe. More iteration space tiling. In *Proceedings of 1989 Conference on Supercomputing*, pages 655–664. ACM Press, November 13–17, 1989.

[25] Michael J. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the 3rd Conference on Parallel Processing for Scientific Computing*, pages 357–361. SIAM Publishers, 1987.

[26] C. H. Wu. A multicolour sor method for the finite-element method. *Journal of Computational and Applied Mathematics*, 30(3):283–294, 1990.