

Combining Performance Aspects of Irregular Gauss-Seidel via Sparse Tiling

Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck
University of California, San Diego
9500 Gilman Dr.
La Jolla, CA 92093-0114
mstrout@cs.ucsd.edu

Abstract

Finite Element problems are often solved using multi-grid techniques. The most time consuming part of multi-grid is the iterative smoother, such as Gauss-Seidel. To improve performance, iterative smoothers can exploit parallelism, intra-iteration data reuse, and inter-iteration data reuse. Current methods for parallelizing Gauss-Seidel on irregular grids, such as multi-coloring and owner-computes based techniques, exploit parallelism and possibly intra-iteration data reuse but not inter-iteration data reuse. Sparse tiling techniques were developed to improve intra-iteration and inter-iteration data locality in iterative smoothers. This paper describes how sparse tiling can additionally provide parallelism. Our results show the effectiveness of Gauss-Seidel parallelized with sparse tiling techniques on shared memory machines, specifically compared to owner-computes based Gauss-Seidel methods. The latter employ only parallelism and intra-iteration locality. Our results support the premise that better performance occurs when all three performance aspects (parallelism, intra-iteration, and inter-iteration data locality) are combined.

1 Introduction

Multigrid methods are frequently used in Finite Element applications to solve simultaneous systems of linear equations. The iterative smoothers used at each of the various levels of multigrid dominate the computation time [3]. In order for iterative smoothers to improve performance, the computation can be scheduled at runtime to exploit three different performance aspects: parallelism, intra-iteration data reuse, and inter-iteration data reuse.

Figure 2 shows the iteration space graph for two commonly-used smoothers, Gauss-Seidel and Jacobi. The iteration space graph in figure 2(a) visually represents the computation and data dependences for the Gauss-Seidel pseudocode in figure 1. The x and y axes suggest that the

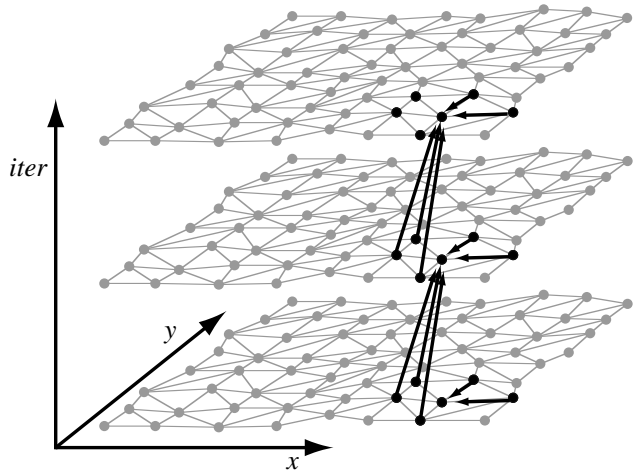
unknowns of the simultaneous equations might lie in a 2-dimensional domain; however, the unknowns are indexed by a single variable i . The $iter$ axis shows three *convergence iterations* (applications of the smoother). Each iteration point $\langle iter, i \rangle$ in the iteration space represents all the computation for the unknown u_i at convergence iteration $iter$. The dark arrows show the data dependences between iteration points for one unknown u_i in the three convergence iterations. At each convergence iteration $iter$ the relationships between the unknowns are shown by the lightly shaded *matrix graph*. Specifically, for each non-zero in the sparse matrix A , $a_{ij} \neq 0$, there is an edge $\langle i, j \rangle$ in the matrix graph.

```
GaussSeidel( $A, \vec{u}, \vec{f}$ )
  for  $iter = 1, 2, \dots, T$ 
    for  $i = 1, 2, \dots, R$ 
       $u_i = f_i$ 
      for all  $j$  in order where ( $a_{ij} \neq 0$  and  $j \neq i$ )
         $u_i = u_i - a_{ij}u_j$ 
       $u_i = u_i/a_{ii}$ 
```

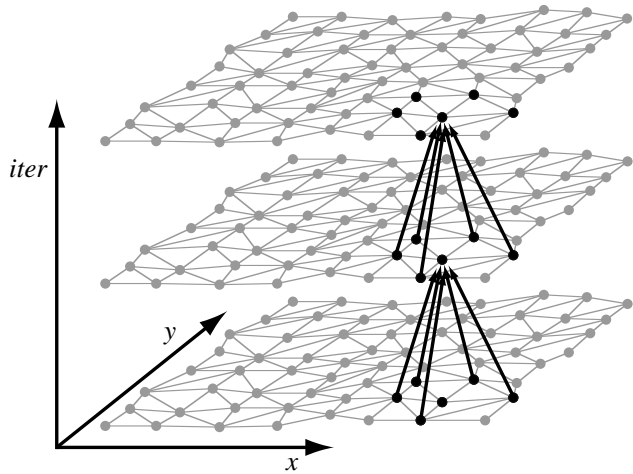
Figure 1. Gauss-Seidel pseudocode

In Gauss-Seidel, each iteration point $\langle iter, i \rangle$ depends on the iteration points of its neighbors in the matrix graph from either the current or the previous convergence iteration, depending on whether the neighbor's index j is ordered before or after i . In Jacobi (figure 2(b)), each iteration point $\langle iter, i \rangle$ depends only on the iteration points of its neighbors in the matrix graph from the previous convergence iteration.

Tiling [35, 20, 14, 34, 8, 24] is a compile-time transformation which subdivides the iteration space for a *regular* computation so that a new tile-based schedule, where each tile is executed atomically, exhibits better data locality. However, sparse matrix computations used for irregular grids have data dependences that are not known un-



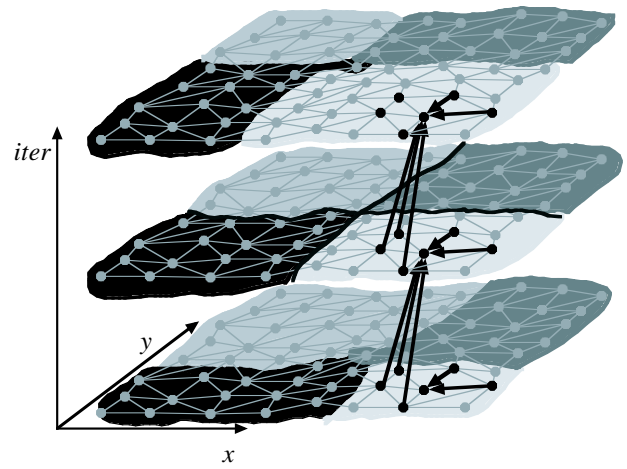
(a) Gauss-Seidel Iteration Space with 3 convergence iterations



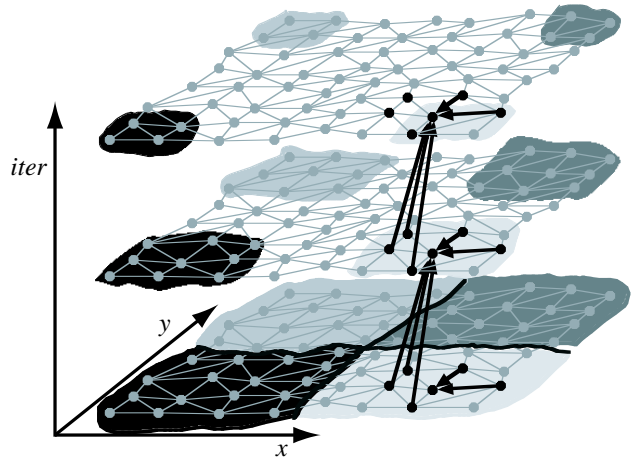
(b) Jacobi Iteration Space with 3 convergence iterations

Figure 2. The arrows show the data dependencies for one unknown u_i . The relationships between the iteration points are shown with a matrix graphs.

til run-time. This prohibits the use of compile-time tiling. Instead, sparse tiling techniques use iteration space slicing [27] combined with inspector-executor [30] ideas to dynamically subdivide iteration spaces induced by the non-zero structure of a sparse matrix (like those shown in figure 2). In the case of Gauss-Seidel, it is necessary to reorder the unknowns to apply sparse tiling. The fact that we can apply an *a priori* reordering requires domain-specific knowledge about Gauss-Seidel.



(a) Sparse tiled Gauss-Seidel iteration space using the full sparse tiling technique. Notice that the seed partition is at the middle convergence iteration.



(b) Sparse tiled Gauss-Seidel iteration space using the cache block sparse tiling technique. The iteration points which are not shaded belong to a tile which will be executed last. The seed partition is at the first convergence iteration.

Figure 3. A visual comparison of the two sparse tiling techniques.

There are two known sparse tiling techniques. Our previous work [33] developed a sparse tiling technique which in this paper we call *full sparse tiling*. Douglas et. al. [12] described another sparse tiling technique which they refer to as cache blocking of unstructured grids. In this paper, we will refer to their technique as *cache block sparse tiling*. Figures 3(a) and 3(b) illustrate how the full sparse

tiling and the cache block sparse tiling techniques divide the Gauss-Seidel iteration space into *tiles*. Executing each tile atomically improves intra- and inter-iteration locality. *Intra-iteration locality* refers to cache locality upon data reuse within a convergence iteration, and *inter-iteration locality* refers to cache locality upon data reuse between convergence iterations.

This paper describes how sparse tiling techniques can also be used to parallelize iterative irregular computations. To parallelize cache block sparse tiled Gauss-Seidel, the “pyramid”-shaped tiles are inset by one layer of iteration points at the first convergence iteration. Then all of the pyramid-shaped tiles can execute in parallel. This however still leaves a large final tile which must be executed serially and which, because of its size, may exhibit poor inter-iteration locality. To parallelize full sparse tiled Gauss-Seidel it is necessary to create a tile dependence graph which indicates the dependences between tiles. Independent tiles can be executed in parallel. We have implemented both sparse tiling techniques within the same framework, therefore, we can compare their effectiveness.

Other methods take advantage of the ability to *a priori* reorder the unknowns in order to parallelize Gauss-Seidel. Multi-coloring is the standard way to parallelize irregular Gauss-Seidel [5]. It works by coloring the matrix graph so that adjacent nodes have different colors. Having done so, all nodes of a given color within one convergence iteration can be executed in parallel. The number of colors is the minimum number of serial steps in the computation. Owner-computes methods use coloring at a coarser granularity. The nodes in the matrix graph are partitioned and assigned to processors. Adjoining partitions have data dependences. Therefore, a coloring of the partition graph can determine which cells in the partitioning can legally be executed in parallel. Adams [2] developed an owner-computes method called nodal Gauss-Seidel, which renumbers the unknowns so that good parallel efficiency is achieved. Both of these techniques require synchronization between convergence iterations.

The main difference between these techniques for parallelizing Gauss-Seidel and sparse tiling techniques is that the former do not directly result in intra-iteration and inter-iteration locality. It is relatively easy to adjust the nodal Gauss-Seidel [2] technique for intra-iteration locality, but neither multi-coloring nor owner-computes based techniques like nodal Gauss-Seidel take advantage of inter-iteration data reuse.

Sparse tiling techniques explicitly manage all three aspects of performance: parallelism, intra-iteration locality, and inter-iteration locality. Although current compilers are not able to analyze a Gauss-Seidel solver and automatically incorporate sparse tiling due to the need for domain specific knowledge, we believe that it will eventually be possible

with the help of user-specified directives.

Section 2 describes sparse tiling from a traditional tiling perspective. In section 3, we describe how to create a parallel schedule for full sparse tiling and show experimental results for parallel executions of full sparse tiled and cache block sparse tiled Gauss-Seidel. In section 4, we qualitatively evaluate methods of parallelizing Gauss-Seidel, including multi-coloring, owner-computes methods, and parallel full sparse tiling, in terms of their intra- and inter-iteration data locality and parallel efficiency. Owner-computes methods are only unable to provide inter-iteration data locality, so we quantitatively compare owner-computes methods with full sparse tiling to investigate the importance of inter-iteration locality. Section 5 discusses future plans for automating sparse tiling techniques, further improving the parallelism exposed by full sparse tiling, and implementing parallel full sparse tiling on a distributed memory machine. Finally, we discuss related work and conclude.

2 A Simple Illustration

Although the focus of this paper is on irregular problems, we would like to introduce our techniques in a simplified setting, using a regular one-dimensional problem as an example.

Suppose we have a vector \vec{u} of N unknowns and want to solve a set of simultaneous equations, $A\vec{u} = \vec{f}$. If the unknowns \vec{u} correspond to some property of points in a suitable one-dimensional physics problem, the matrix A will be tri-diagonal, i.e. the only non-zeros will be in the major diagonal and the two adjacent diagonal. In this case, the following code corresponds to applying three iterations of a Jacobi smoother (assuming we have initialized $u[0]=u[N+1]=0$):

```

for iter = 1 to 3
  for i = 1 to N
    newu[i] = (f[i]-A[i,i-1]*u[i-1]
              -A[i,i+1]*u[i+1]) / A[i,i]
  for i = 1 to N
    u[i] = newu[i]

```

Under certain assumptions about the matrix A , after each iteration of the *iter* loop, \vec{u} will be a closer approximation to the solution of the simultaneous equations (hence the term, “convergence iterations”).

Our goal is to parallelize this computation and to improve the use of the computer’s memory hierarchy through intra- and inter-iteration locality. The simplest method of parallelizing it for a shared-memory computer is to partition the \vec{u} and *newu* vectors among the processors. Then for each convergence iteration, each processor can compute

its portion of $ne\vec{w}u$ in parallel. Next, the processors perform a global synchronization, copy their portion of $ne\vec{w}u$ to \vec{u} , resynchronize, and proceed to the next convergence iteration. On a distributed memory machine, the two synchronizations are replaced by a single communication step.

In this example, the resulting code will have decent intra-iteration locality. Specifically, the elements of \vec{u} , $ne\vec{w}u$, and A are accessed sequentially (i.e., with spatial locality). Further, in the i loop, each element $u[i]$ is used when calculating $u[i-1]$ and $u[i+1]$, which results in temporal locality.

However, there is no inter-iteration locality. During each convergence iteration, each processor makes a sweep over its entire portion of A , \vec{u} and $ne\vec{w}u$, and then a second sweep over \vec{u} and $ne\vec{w}u$. Since these data structures are typically much larger than caches, there is no temporal locality between convergence iterations. Such locality is important because even with prefetching, most processors cannot fetch data from memory as fast as they can perform arithmetic.

Tiling provides a method of achieving inter-iteration locality. The rectangle in figure 4 represents the $3 \times N$ iteration space for the example code - each point in the rectangle represents a computation of $newu[i]$. To simplify the exposition, we ignore the storage-related dependences and the copy of $ne\vec{w}u$ to \vec{u} , but note in passing that this is a sophisticated application of tiling involving two loops, that neither tile boundary is parallel to the iteration space boundary, and that the tiling could benefit from the storage-savings techniques of [33].

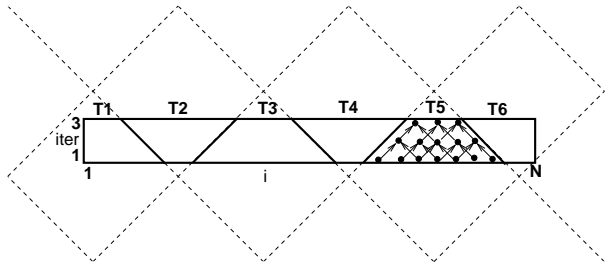


Figure 4. Tiling the two-dimensional iteration space corresponding to a simple one-dimensional regular stencil computation. The dashed lines illustrate how the trapezoidal tiles arise from a regular parallelogram tiling of a conceptually infinite space. The arrows in tile T5 show the dependences between iterations.

In figure 4, there are six tiles labeled T1 to T6. For any of the odd-numbered tiles, all of the computations in the tile can be executed without needing the results from any

other tiles. After all the odd-numbered tiles are completed, then the even-numbered tiles can be executed in parallel. Furthermore, assuming the tiles are made sufficiently small, each tile's portions of the three arrays will remain in cache during the execution of the tile. Thus, tiling can achieve parallelism as well as intra- and inter-iteration locality for a simple regular computation. Unfortunately, tiling requires that the dependences be regular. If we replace the tridiagonal matrix A by an arbitrary sparse matrix then the implementation will use indirect memory references and non-affine loop bounds and tiling will no longer be applicable.

We now illustrate how sparse tiling achieves parallelism, as well as intra- and inter-iteration locality. Sparse tiling is a run-time technique that partitions the $ne\vec{w}u$ vector into cells that can conveniently fit into cache, then chooses an order on the cells, and then "grows" each cell into the largest region of the iteration space that can be computed consistent with the dependences and the ordering on the cells. The tiles are grown so that each tile can be executed atomically. Since all of this is done at runtime, sparse tiling is only profitable if the overhead of forming the tiles can be amortized over multiple applications of the smoother on the same underlying matrix. Fortunately, this is often the case.

Figure 5 shows how sparse tiling would work on our simple example. The horizontal axis (representing the indices of the \vec{u} and $ne\vec{w}u$ vectors) is partitioned into six cells. In the top diagram, these are numbered sequentially from left to right; in the bottom, they are given a different numbering. Then, each tile in turn is "grown" upwards assuming adjacent tiles will be executed in numerical order. If the neighboring iteration points have already been computed by an earlier tile, then the upward growth can expand outwards; otherwise, the tile contracts as it grows upwards through the iteration space.

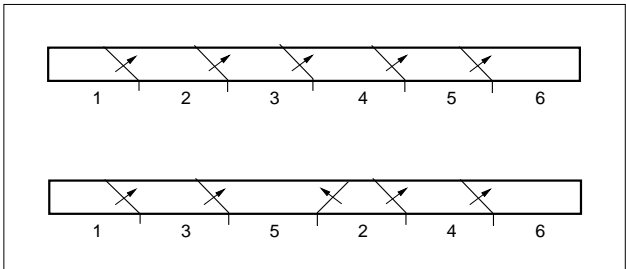


Figure 5. Two applications of sparse tiling to the iteration space of figure 4. In both cases, the data was partitioned into six cells. In the top diagram, they were ordered from left to right. The resulting tiling achieves inter-iteration locality. The small arrows indicate dependences between tiles.

Both diagrams result in a tiling that achieves inter-iteration locality. However, there will be dependences between the tiles, as shown by the arrows in the diagrams. The six tiles in the top diagram must be executed sequentially — tile $i + 1$ cannot be executed until tile i is completed, under the usual assumption that tiles are executed atomically. However, in the bottom diagram, the tiles numbered 1 and 2 can be executed concurrently. When 1 is complete, 3 can be started; when 2 is done, 4 can be started, and so on. Thus, sparse tiling can achieve parallelism, intra-iteration locality, and inter-iteration locality on irregular problems. However, it requires either luck or a thoughtful choice for the initial numbering of the cells given by the partitioner.

In the remainder of this paper, we will move away from the simplifying assumptions of this section. In particular, we will consider higher dimensional, unstructured problems. Thus, there will be more non-zeros in the A matrix, they will occur in an irregular pattern, and a space-efficient sparse data structure will be used. Furthermore, we will concentrate on using a Gauss-Seidel smoother rather than a Jacobi one. This eliminates the temporary vector $ne\vec{w}u$. Instead, each i iteration reads and updates the \vec{u} vector which introduces dependences within a convergence iteration as well as between convergence iterations. As mentioned earlier, it is common with Gauss-Seidel that a reordering of the unknowns is permitted, provided that once an order is chosen, the same order is used throughout the execution of the iteration space. This allows us to choose any numbering on the cells of the partitioning as well.

3 Executing Sparse Tiled Gauss-Seidel in Parallel

Sparse tiling techniques perform runtime rescheduling and data reordering by partitioning the matrix graph, growing tiles from the cells of the seed partitioning, constructing the new data order, and creating the new schedule based on the sparse tiling. In order to execute tiles in parallel we construct a tile dependence graph. The tile dependence graph is used by a master-worker implementation. The master puts tiles whose data dependences are satisfied on a ready queue. The workers execute tiles from the ready queue and notify the master upon completion.

The following is an outline of the sparse tiling process for parallelism.

- **Partition** the matrix graph to create the seed partitioning. Each piece of the partition is called a *cell*. Currently we use the Metis [22] partitioning package for this step.
- **Choose a numbering** on the cells of the seed partition.
- **Grow tiles** from each cell of the seed partitioning in

Name	Description	L2 cache
Ultra	SUN HPC10000, up to 32 400 MHz UltraSPARCII processors	4MB
Blue Horizon Node	One node of an IBM SP, Eight 375 MHz Power3 processors	8MB

Table 1. Descriptions of architectures used in experiments.

turn to create the tiling function θ which assigns each iteration point to a tile. The tile growth algorithm will also generate constraints on the data reordering function.

- **Reorder** the data using the reordering function.
- **Reschedule** by creating a schedule function based on the tiling function θ . The schedule function provides a list of iteration points to execute for each tile at each convergence iteration.
- **Generate tile dependence graph** identifying which tiles may be executed in parallel.

Either full tile growth (called serial sparse tiling in [33]) or cache blocking tile growth [12] can be used to grow tiles based on an initial matrix graph partitioning. We show results for both methods.

Our experiments were conducted using the IBM Blue Horizon and SUN Ultra at the San Diego Supercomputer center. Details on both machines are given in table 1. We generated three matrices by solving a linear elasticity problem on a 2D bar, a 3D bar, and a 3D pipe using the FEtk [18] software package. The Sphere and Wing examples are provided by Mark Adams [1]. These are also linear elasticity problems.

In all our experiments, the sparse matrix is stored in a compressed sparse row (CSR) format. Our previous work [33] compared sparse tiled Gauss-Seidel (with a serial schedule) using CSR with a version of Gauss-Seidel which used a blocked sparse matrix format with a different format for the diagonal blocks, upper triangle blocks, and lower triangle blocks. The blocked sparse matrix format exploits the symmetric nature of the sparse matrices generated by the Finite Element package FEtk [18]. Further experimentation has shown that the CSR matrix format results in a more efficient implementation of the typical Gauss-Seidel schedule.

Our experiments examine the raw speedup of sparse tiled versions of Gauss-Seidel over a typical schedule for Gauss-Seidel (as shown in figure 1), the overhead of computing

Matrix	numrow	num non-zeros	avg non-zeros per row
2D Bar	74,926	1,037,676	13.85
3D Bar	122,061	4,828,779	39.56
Sphere150K	154,938	11,508,390	74.28
Pipe	381,120	15,300,288	40.15
Wing903K	924,672	38,360,266	41.49

Table 2. Descriptions of input matrices.

the sparse tiling, and the average parallelism within the tile dependence graph.

3.1 Raw Speedup

Figure 6 shows the raw speedups of cache blocked and full sparse tiled Gauss-Seidel for 1, 2, 4, and 8 processors on a node of the IBM Blue Horizon. Figure 7 shows the raw speedups on the SUN Ultra using up to 32 processors. In [12], cache block sparse tiling uses seed partitions which fit into half of L2 cache, and we do the same. In our experience, full sparse tiling gets better performance when we select the seed partition size to fit into one-eighth of L2 cache. More work is needed to improve on this heuristic.

While both sparse tiling techniques achieve speedups over the unoptimized Gauss-Seidel, full sparse tiling frequently achieves the best speedups. With cache block sparse tiling all of the tiles are executed in parallel except for the last tile. This last tile cannot be started until all the other tiles have completed, so parallelism is inhibited. Further, the last tile may be large and therefore have poor inter-iteration locality and intra-iteration locality.

Sparse tiling performs well in all cases but the Wing matrix on the Ultra. Further investigation is needed to determine the problem with the largest problem set.

3.2 Overhead

Sparse tiling techniques are performed at runtime, therefore the overhead of performing sparse tiling must be considered. We present the overhead separately because Gauss-Seidel is typically called many times within applications like multigrid. We can amortize the overhead over these multiple calls which use the same sparse matrix. Our results show that Gauss-Seidel with two convergence iterations must be called anywhere from 56 to 194 times on the sample problems to amortize the overhead. Specific break even points are given in table 3. On average, 75% of the overhead is due to the graph partitioner Metis. A breakdown of the overhead per input matrix is given in table 4.

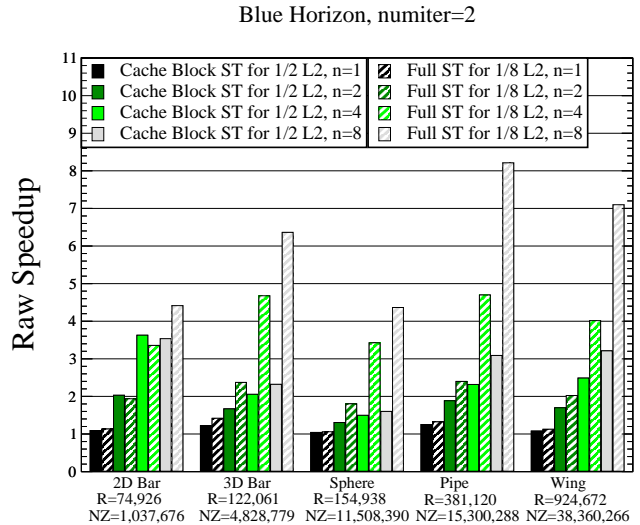


Figure 6. Raw speedup of sparse tiled Gauss-Seidel with 2 convergence iterations over a typical Gauss-Seidel schedule. These experiments were run on a node of the IBM Blue Horizon at SDSC. Each node has 8 Power3 processors.

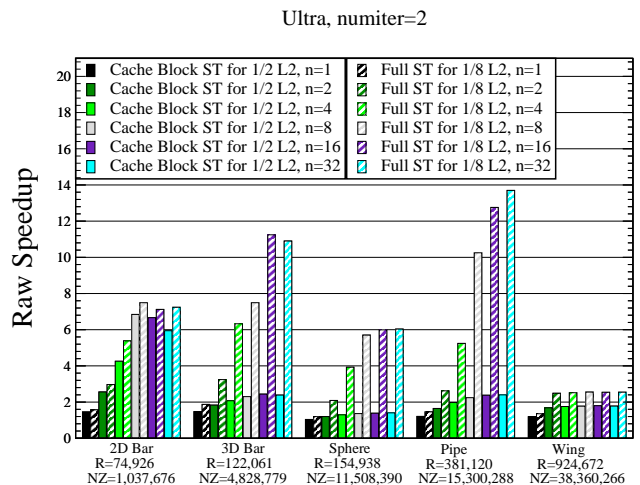


Figure 7. Raw speedup of sparse tiled Gauss-Seidel with 2 convergence iterations over a typical Gauss-Seidel schedule. These experiments were run on a SUN HPC10000 which has 36 UltraSPARCII processors with uniform memory access.

Blue Horizon, Gauss-Seidel with numiter=2, Rescheduling for parallelism							
Input Matrix	Overhead(sec)	Savings/Execution (sec)			Break Even (# executions)		
		n=2	n=4	n=8	n=2	n=4	n=8
Matrix9	2.03	0.02	0.03	0.04	90	63	57
Matrix12	13.69	0.14	0.20	0.21	96	71	66
Sphere150K	31.41	0.17	0.26	0.29	191	120	110
PipeOT15mill	48.28	0.39	0.52	0.58	125	93	83
Wing903K	116.86	0.65	0.96	1.10	182	122	107

Table 3. Number of Gauss-Seidel (2 convergence iterations) executions required to amortize sparse tiling overhead.

Blue Horizon, Gauss-Seidel with numiter=2, Rescheduling for parallelism		
Input Matrix	Partition Time	Data Reordering
Matrix9	78.92%	14.06%
Matrix12	71.89%	13.42%
Sphere150K	67.64%	16.53%
PipeOT15mill	81.42%	9.73%
Wing903K	83.58%	9.95%

Table 4. Break down of the overhead time. Around 80% to 90% of the overhead is due to partitioning the matrix graph plus reordering the unknown vector, right-hand side, and sparse matrix.

Owner computes parallelization methods for sparse matrices also require a partitioner and data reordering is necessary for parallelizing Gauss-Seidel.

It is possible to reduce the overhead by using faster matrix graph partitioners and by reducing the size of the matrix graph. The results in this paper use the `Metis_PartGraphRecursive` function for the matrix graph partitioning. Preliminary experiments show that the `Metis_PartGraphKway` function is much faster, and the resulting raw speedups decrease only slightly. We are also experimenting with the `GPART` partitioner [16].

Previous sparse tiling work [33, 12] performed full sparse tiling and cache block sparse tiling on the input mesh, instead of the resulting matrix graph. Since there are often multiple unknowns per mesh node in a finite element problem, the resulting matrix graph will have multiple rows with the same non-zero structure. In such cases, the mesh will be d^2 times smaller than the resulting sparse matrix, where d is the number of unknowns per mesh node. Future work will consider compressing general matrix graphs by discovering rows with the same non-zero structure.

3.3 Increasing Parallelism with Graph Coloring

The degree of parallelism within sparse tiled Gauss-Seidel is a function of the tile dependence graph. Specif-

ically, the height of the tile dependence graph indicates the critical path of the computation. A more useful metric in determining the amount of parallelism available is the total number of tiles divided by the height of the tile dependence graph, which we refer to as the average parallelism.

For example, figure 5 gives two sparse tilings of the same iteration space graph. The tile dependence graphs for those sparse tilings are shown in figure 8. The first tile dependence graph, which exhibits no parallelism, has height equal to 6 and average parallelism equal to 1. The second tile dependence graph has height 3 and average parallelism 2. Therefore, the second sparse tiling has enough parallelism to keep two processors busy, assuming that each tile requires roughly the same amount of computation time.

Potentially we can execute the second sparse tiling twice as fast. The two sparse tilings differ in their original numbering of the cells of the seed partition. The tile growth algorithms use that numbering to indicate the execution order for adjacent tiles, thus the partition numbering affects the data dependence direction between tiles.

We compared the partition numbering provided by `Metis`, a random numbering, and a numbering based on a coloring of the partition graph. The partition graph is an undirected graph with a node for each cell of the matrix graph partitioning. When cells A and B share an edge or multiple edges in the matrix graph, there is an edge (A, B)

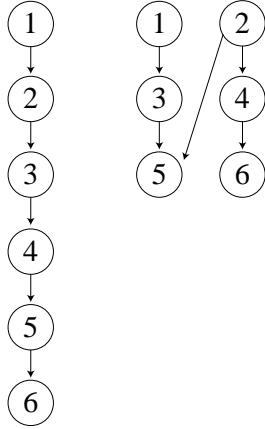


Figure 8. Tile dependence graphs for the sparse tilings shown in figure 5. Each circle represents a tile, and arrows represent data flow dependences. For example, in the first tile dependence graph, tile 1 must execute before tile 2.

in the partition graph. We color the nodes of the partition graph, and then assign consecutive numbers to the cells of the partitioning which correspond to nodes of a given color. This way, the tiles grown from the cells of a given color will probably not be data dependent. After tile growth, the data dependences between tiles must be calculated to insure correctness, since even though two partition cells are not adjacent, the tiles grown from the cells may be dependent. In our experiments, we use the greedy heuristic provided in the Graph Coloring Programs [10] to color the partition graph.

The graph in figure 9 shows the average parallelism for four different matrices with full sparse tiling using cells that fit into one eighth of an 8 MB L2 cache. Using graph coloring on the partition graph uniformly improves the degree of parallelism.

The importance of the average parallelism in the tile dependence graph can be seen when we examine the raw speedup of full sparse tiled Gauss-Seidel using the three different partition numberings. In figure 10, notice that the top two lines with speedups for the Pipe matrix show nearly linear speedup, corresponding to the fact that average parallelism for the Pipe matrix is over 16 for a random partition numbering and a graph coloring based partition numbering. However, the speedup is much less than linear when the number of processors is larger than the average parallelism in the tile dependence graph, as illustrated by the other four lines of figure 10.

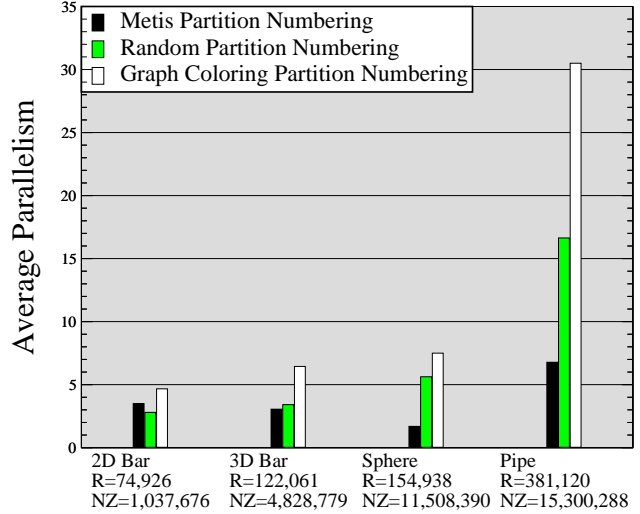


Figure 9. The average parallelism in the tile dependence graph for full sparse tiled Gauss-Seidel with 2 convergence iterations.

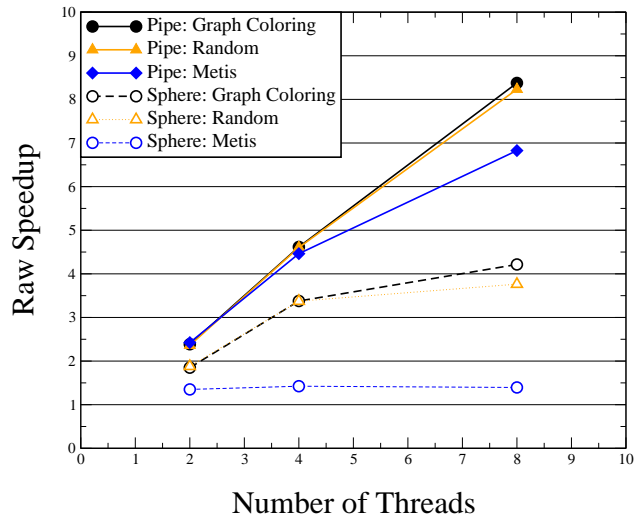


Figure 10. The effect that average parallelism has on speedup for full sparse tiled Gauss-Seidel with 2 convergence iterations.

4 Comparison with other Gauss-Seidel Parallelization Techniques

Sparse tiling techniques differ from other Gauss-Seidel parallelization techniques, specifically multicoloring and owner-computes methods, in their focus on improving intra- and inter-iteration locality. Since in all these parallelization methods each processor is given an approximately equal

	Parallel Efficiency	Intra-iteration locality	Inter-iteration locality
Multi-coloring	yes	no	no
Owner-computes	yes	yes	no
Sparse tiling	yes	yes	yes

Table 5. Summary of how the various Gauss-Seidel parallelization techniques compare in how they handle the three performance aspects.

amount of work, less than linear speedup may be due to parallel inefficiencies and/or poor data locality. In this section we compare the parallel efficiency, intra-iteration locality, and inter-iteration locality of multi-coloring, owner-computes methods, and sparse tiling techniques.

4.1 Parallel Efficiency

In shared memory parallel processing, the synchronization time is the amount of time that processors are waiting for data dependent results that are generated by other processors. Parallel efficiency occurs when the synchronization time is minimized. For owner-computes parallelized Gauss-Seidel there is intra-iteration synchronization because adjacent cells of the matrix graph partitioning will depend on each other. Nodal Gauss-Seidel reorders the unknowns so that intra-iteration synchronization is hidden and therefore parallel efficiency is maintained.

For multi-coloring and owner-computes methods, as long as each processor is given approximately the same number of unknowns and associated matrix rows, the synchronization barrier between convergence iterations will not cause much parallel inefficiency.

Because they group multiple convergence iterations together, sparse tiling techniques only have synchronization issues between tiles, instead of intra-iteration and inter-iteration synchronization. As long as the tile dependence graph has enough parallelism to feed the available processors, full sparse tiled Gauss-Seidel should have good parallel efficiency.

4.2 Intra-iteration locality

Multi-coloring techniques have poor intra-iteration locality because in order for iteration point $\langle iter, v \rangle$ to be executed in parallel with other iteration points, $\langle iter, v \rangle$ must not be a neighbor of the other iteration points. However, neighboring iteration points reuse the same data.

When executing many iteration points that are not neighbors, data reuse is not local.

Owner-computes methods like nodal Gauss-Seidel can easily improve their intra-iteration locality by further partitioning the sub-matrix on each processor, and reordering the unknowns based on that partitioning [16].

The partitions used to grow sparse tiles are selected to be small enough to fit into (some level of) cache. Therefore the data reordering will result in intra-iteration locality.

4.3 Inter-iteration locality

Both multicolored and owner computes Gauss-Seidel execute all the iteration points within one convergence iteration before continuing to the next convergence iteration. If the subset of unknowns (and their associated sparse matrix rows) assigned to a processor do not fit into a level of cache then no inter-iteration locality occurs.

Sparse tiling techniques subdivide the iteration space so that multiple convergence iterations over a subset of the unknowns occur atomically, thus improving the inter-iteration locality.

4.4 Experimental Comparison

Since owner-computes methods differ from sparse tiling methods only by their lack of inter-iteration locality, we compare the two by simulating an owner-computes method. We refer to the experiment as a simulation because the Gauss-Seidel dependences are violated in order to give the owner-computes method perfect intra-iteration parallel efficiency. This simulates the performance of a complete Nodal Gauss-Seidel implementation which has good intra-iteration parallel efficiency. Inter-iteration parallel efficiency within the owner-computes simulation is achieved by giving each processor the same number of unknowns. Finally, intra-iteration locality is provided by partitioning the sub-matrix graph on each processor and then reordering the unknowns accordingly.

The Sphere, Pipe, and Wing problems are the only data sets that don't fit into L2 cache once the data is divided up for parallelism. The Sphere matrix has an average number of non-zeros per row of 74.28 (as shown in table 2). This causes sparse tiles to grow rapidly and therefore results in poor parallelism in the tile dependence graph. Recall in figure 9 that the maximum average parallelism was 7.5 for 2 convergence iterations when tiled for the Blue Horizon's L2 caches. This average parallelism worsens to 3.2 for 4 convergence iterations. The lack of parallelism causes poor performance in the full sparse tiled Gauss-Seidel on the Blue Horizon for the Sphere dataset (figure 11). However, with the Pipe and Wing matrices the average number of non-zeros per row are much lower at 40.15 and 41.49. Cor-

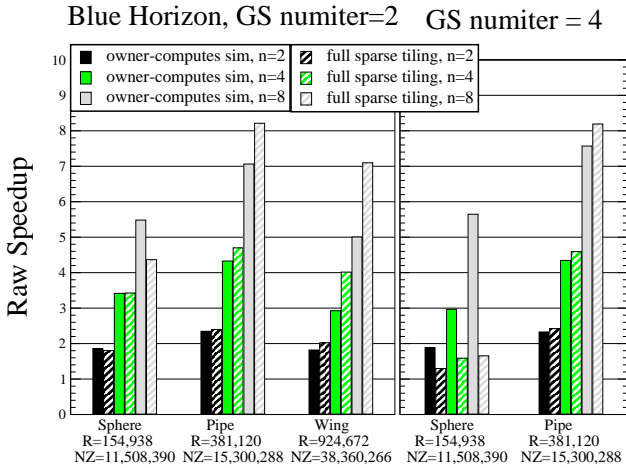


Figure 11. Full sparse tiled Gauss-Seidel with 2 and 4 convergence iterations compared with the owner computes simulation. These experiments were run on one node of the IBM Blue Horizon at SDSC.

respondingly, the average parallelism when tiling for one-eighth of the Blue Horizon L2 cache is 30.5 and 56.6 for 2 convergence iterations. Therefore for 2, 4, or 8 processors there is plenty of parallelism for the Pipe and Wing problems on the Blue Horizon.

On the Ultra (results shown in figure ??), the L2 cache is smaller so more tiles were used to fit into one-eighth of the L2 cache. This increased the average parallelism for the Sphere problem to 9.3 for Gauss-Seidel with 2 convergence iterations. The speedup on the Ultra for the Sphere problem is maximized around 6 even though there is more parallelism available. When sparse tiling Sphere Gauss-Seidel for 3 convergence iterations the average parallelism for Sphere reduces to 4.66, and the full sparse tiled speedup never hits 3. The owner computes simulation outperforms full sparse tiling in this instance, because in this one case full sparse tiling doesn't generate enough parallelism.

The Wing results on the Ultra are curious. The tile dependence graph for 2 convergence iterations has 90.5 average parallelism and for 3 convergence iterations has 82.3 average parallelism. However, even though the Wing problem has been rescheduled for parallelism, inter-iteration locality, and intra-iteration locality, the speedup never breaks 4. We conjecture that this is due to the size of the problem and possible limits on the Ultra.

Our experiments show that as long as the tile dependence graph generated by full sparse tiling has enough parallelism, full sparse tiled Gauss-Seidel outperforms owner computes methods on shared memory architectures. Our owner computes simulation made idealized assumptions about the

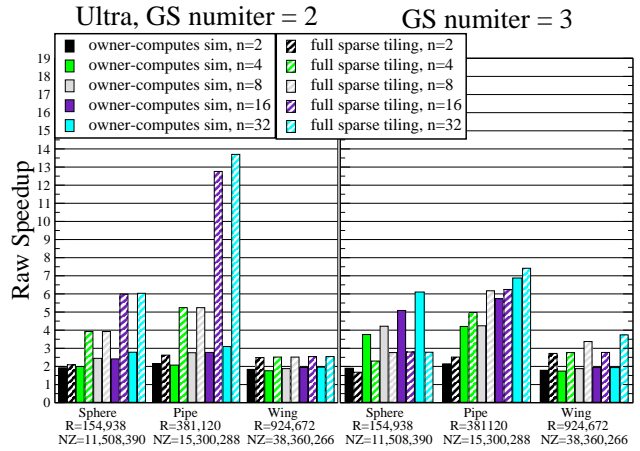


Figure 12. Full sparse tiled Gauss-Seidel with 2 and 3 convergence iterations compared with the owner computes simulation. These experiments were run on the SUN HPC10000 at SDSC.

intra-iteration parallel efficiency and added intra-iteration locality. These results show that inter-iteration locality is an important performance aspect that owner computes methods are missing.

5 Future Work

Automating the use of sparse tiling techniques is an important next step to increase the usefulness of such techniques. Sparse tiling techniques use domain-specific information in order to reschedule Gauss-Seidel for parallelism, intra-iteration locality, and inter-iteration locality. Currently a number of research projects are exploring ways of optimizing the use of domain-specific libraries. Sparse tiling is most applicable to libraries which contain a large amount of sparse matrix iterative computations.

ROSE [28] is a system which generates domain-specific preprocessors. Their framework supports translating the general abstract syntax tree (AST) to a higher-level domain-specific AST, on which transformations for performance optimizations can then be performed. Interface compilation [13] and telescoping languages [9] also look at ways of optimizing uses of library interfaces. Others have looked at the specific example of compiler transformations for Matlab which is a domain-specific language [4]. Being able to attach domain-specific semantics to the library interface would allow us to construct a preprocessor which recognizes that the unknown vector being passed to a Gauss-Seidel function may be *a priori* reordered.

The Broadway compiler [7] allows the library expert

to specify annotations for domain-specific, higher-level dataflow analysis. We can apply these ideas to determine what other data structures will be affected by doing an *a priori* reordering of the unknown vector in a Gauss-Seidel invocation.

Another important step for sparse tiling techniques will be the ability to run on distributed memory machines. This will require calculating the data footprint of all the tiles and creating an allocation of tiles to processors which results in parallel efficiency.

Finally, currently sparse matrices with a large ratio of non-zeros to rows result in tiles with many more dependences than the original cells of the seed partitioning. It might be possible to add edges to the partition graph before coloring it, so that the final tile dependence graph will have fewer dependences.

6 Related Work

Both iteration space slicing [27] and data shackling [23] are techniques which divide up the iteration space based on an initial data partition. This is exactly what sparse tiling does, but sparse tiling handles irregular iteration space graphs, whereas iteration space slicing and data shackling are applicable in loops with affine loop bounds and array references.

Since the smoother dominates the computation time in multigrid methods, much work revolves around parallelizing the smoother. This paper focuses on parallelizing an existing iterative algorithm with good convergence properties, Gauss-Seidel. Another approach is to use smoothers which are easily parallelizable like domain decomposition [31], blocked Jacobi, or blocked Gauss-Seidel [17]. Relative to Gauss-Seidel these approaches have less favorable convergence properties. For example, the convergence rate depends on the number of processors and degrades as this number increases [15].

There has also been work on run-time techniques for improving the intra-iteration locality for irregular grids which applies a data reordering and computation rescheduling within a single convergence iteration [25, 26, 11, 19, 16]. We use graph partitioning of the sub-matrices to give our owner-computes simulation better intra-iteration locality. However, many of these techniques do not apply to Gauss-Seidel because it has data dependences within the convergence iteration.

Work which looks at inter-iteration locality on regular grids includes [6], [32], [29], [21], and [36]. The only other technique to our knowledge which handles inter-iteration locality for irregular meshes is unstructured cache-blocking by Douglas et al.[12]. We have implemented this technique in our experimental framework and refer to it as cache block sparse tiling in this paper.

7 Conclusion

Sparse tiling explicitly creates intra-iteration locality, inter-iteration locality, and parallelism for irregular Gauss-Seidel. The combination of these three performance aspects results in high performance. This paper describes how full sparse tiling can be used to parallelize Gauss-Seidel by creating a tile dependence graph. Full sparse-tiled Gauss-Seidel is compared with an owner-computes based parallelization, and when all aspects of performance are available, sparse tiled Gauss-Seidel has better speedups, due to the lack of inter-iteration locality in owner-computes based methods.

8 Acknowledgments

This work was supported by an AT&T Labs Graduate Research Fellowship, a Lawrence Livermore National Labs LLNL grant, and in part by NSF Grant CCR-9808946. Equipment used in this research was supported in part by the UCSD Active Web Project, NSF Research Infrastructure Grant Number 9802219 and also by the National Partnership for Computational Infrastructure (NPACI).

We would like to thank Professor Mike Holst for his assistance with the FETk software package and general information about Finite Element Analysis. We would also like to thank the reviewers for comments which helped improve the paper.

References

- [1] Mark F. Adams. Finite element market. <http://www.cs.berkeley.edu/~madams/femarket/index.html>.
- [2] Mark F. Adams. A distributed memory unstructured Gauss-Seidel algorithm for multigrid smoothers. In ACM, editor, *SC2001: High Performance Networking and Computing*. Denver, CO, 2001.
- [3] Mark F. Adams. Evaluation of three unstructured multigrid methods on 3D finite element problems in solid mechanics. *International Journal for Numerical Methods in Engineering*, To Appear.
- [4] George Almšić and David Padua. Majic: Compiling matlab for speed and responsiveness. In *PLDI 2002*, 2002.
- [5] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

- [6] F. Basseti, K. Davis, and D. Quinlan. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. *Lecture Notes in Computer Science*, 1505, 1998.
- [7] Emergy Berger, Calvin Lin, and Samuel Z. Guyer. Customizing software libraries for performance portability. In *10th SIAM Conference on Parallel Processing for Scientific Computing*, March 2001.
- [8] Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. *The Journal of Supercomputing*, pages 114–124, November 1992.
- [9] A. Chauhan and K. Kennedy. Optimizing strategies for telescoping languages: Procedure strength reduction and procedure vectorization. In *Proceedings of the 15th ACM International Conference on Supercomputing*, pages 92–102, New York, 2001.
- [10] Joseph Culberson. Graph coloring programs. <http://www.cs.ualberta.ca/~joe/Coloring/Colorsrc/index.html>.
- [11] 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.
- [12] 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.
- [13] Dawson R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, May/June 1999.
- [14] 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.
- [15] M.J. Hagger. Automatic domain decomposition on unstructured grids (doug). *Advances in Computational Mathematics*, (9):281–310, 1998.
- [16] Hwansoo Han and Chau-Wen Tseng. A comparison of locality transformations for irregular codes. In *5th International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR'2000)*. Springer, 2000.
- [17] Van Emden Henson and Ulrike Meier Yang. Boomer-AMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics: Transactions of IMACS*, 41(1):155–177, 2002.
- [18] Michael Holst. Fetk - the finite element tool kit. <http://www.fetk.org>.
- [19] Eun-Jin Im. *Optimizing the Performance of Sparse Matrix-Vector Multiply*. Ph.d. thesis, University of California, Berkeley, May 2000.
- [20] F. Irigoien and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 319–329, 1988.
- [21] Guohua Jin, John Mellor-Crummey, and Robert Fowler. Increasing temporal locality with skewing and recursive blocking. In ACM, editor, *SC2001: High Performance Networking and Computing*. Denver, CO, November 10–16, 2001. ACM Press and IEEE Computer Society Press, 2001.
- [22] 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.
- [23] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 346–357, New York, June 15–18 1997. ACM Press.
- [24] 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.
- [25] 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, June 1999.
- [26] 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.
- [27] William Pugh and Evan Rosser. Iteration space slicing for locality. In *LCPC Workshop*, La Jolla, California, August 1999. LCPC99 website.

- [28] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. In *Proceedings of Conference on Parallel Compilers (CPC2000)*, Aussois, France, January 2000. Also published in a special issue of *Parallel Processing Letters*, Vol.10.
- [29] 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.
- [30] Shamik D Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and compile-time support for adaptive irregular problems. In *Supercomputing '94*. IEEE Computer Society, 1994.
- [31] Barry F. Smith, Petter E. Bjørstad, and William Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.
- [32] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices*, 34(5):215–228, May 1999.
- [33] 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, New Haven, Connecticut, May 28-30, 2001. Springer.
- [34] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation*, 1991.
- [35] Michael J. Wolfe. Iteration space tiling for memory hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1987.
- [36] David Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):181, June 2002.