UNIVERSITY OF CALIFORNIA, SAN DIEGO

Performance Transformations for Irregular Applications

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science and Engineering

by

Michelle Mills Strout

Committee in charge:

Professor Larry Carter, Co-Chair
Professor Jeanne Ferrante, Co-Chair
Professor William Griswold
Professor Michael Holst
Professor Andrew McCulloch

2003
The dissertation of Michelle Mills Strout is approved, and it is acceptable in quality and form for publication on microfilm:

Co-Chair

Co-Chair

University of California, San Diego

2003
To my husband Joe
TABLE OF CONTENTS

Signature Page .................................................. iii
Dedication Page .................................................. iv
Table of Contents ................................................ v
List of Figures .................................................. viii
List of Tables .................................................. xiii
Acknowledgments ................................................ xv
Vita, Publications, and Fields of Study ....................... xvii
Abstract .......................................................... xix

1 Introduction .................................................. 1
   1. Background .............................................. 4
   2. Sparse Tiling for Stationary Iterative Methods ........... 10
   3. Using Sparse Tiling in Existing Software Packages ....... 14
   4. Run-time Reordering Transformation Framework ........... 16
   5. Contributions ........................................... 18
   6. Dissertation Overview ................................... 19

2 Sparse Tiling Stationary Iterative Methods for Locality .... 20
   1. Visualizing Stationary Iterative Methods ................. 21
   2. Visualizing Sparse Tiling for Gauss-Seidel ............... 23
      1. Data Dependence Relations in Gauss-Seidel for CSR ...... 28
      2. Compile-time Transformation Mappings ................. 32
      3. Constraints on the Tiling Function Due to Data Dependences . 34
   4. Sparse Tiling Inspector for Gauss-Seidel ................. 38
      1. Partition the Matrix Graph .......................... 39
      2. Sparse Tile the Iteration Space ....................... 40
      3. Generate the Reordering Function .................... 53
      4. Remap the Data ...................................... 54
      5. Create Schedule .................................... 54
      6. Proof of Correctness ............................... 54
      7. Full Sparse Tiling Efficiency Issues .................. 56
   5. Matrix Benchmark Suite ................................ 63
6. Experimental Results for Locality ........................................ 65
1. Partition Size Selection ............................................... 66
2. Number of Convergence Iterations ............................... 71
3. Full Sparse-Tiled Gauss-Seidel vs. Cache-Blocked Gauss-Seidel 73
4. Overhead ................................................................. 74
7. Sparse Tiling other Stationary Iterative Methods .......... 78
8. Related Work ............................................................... 80
9. Summary ................................................................. 82

3 Sparse Tiling Stationary Iterative Methods for Parallelism ...... 84
1. Executing Sparse-Tiled Gauss-Seidel in Parallel ........... 84
   1. Increasing Parallelism with Graph Coloring .......... 86
   2. Constructing the Tile Dependence Graph ............. 90
   3. Parallel Executor Speedup ........................... 93
   4. Overhead for Generating a Parallel Schedule ......... 95
2. Comparison with other Gauss-Seidel Parallelization Transformations 97
   1. Parallel Efficiency .................................. 98
   2. Intra-iteration Locality ................................ 99
   3. Inter-iteration Locality ................................ 99
   4. Experimental Comparison .......................... 100
3. Related Work ......................................................... 103
4. Summary ................................................................. 105

4 Using Sparse Tiling in Existing Software Packages .......... 106
1. Sparse Tiling in FETk ............................................. 107
   1. Interface between Finite Element Analysis and Multigrid 107
   2. Using Sparse Tiling in Symmetric Multigrid .......... 112
   3. Using FETk to Generate the Matrix Hierarchy ...... 116
   4. Experimental Results with Symmetric Multigrid .... 117
2. Sparse Tiling in PETSc ............................................ 122
   1. PETSc Overview ....................................... 123
   2. Possibilities for Sparse Tiling SOR .................. 125
   3. STPetsc Two-Level Interface .......................... 129
   4. Experimental Results Using STPetsc ................ 131
3. Related Work ......................................................... 138
4. Summary ................................................................. 142

5 Run-time Reordering Transformation Framework ............ 143
1. Motivation for Composing Run-time Reordering
   Transformations ....................................................... 144
   1. Run-time Data Reordering Transformations .......... 145
LIST OF FIGURES

1.1 Ideally the compiler would automatically generate the inspector/executor code and any necessary decision code to implement run-time reordering transformations. ........................................ 3
1.2 Simple irregular loop with one indirect memory reference . . . 4
1.3 The data mapping and associated cache behavior for the original computation. The circles represent iterations in the loop and the M’s represent cache misses due to the memory reference into the data array Z. .......................................................... 5
1.4 The data mapping and associated cache behavior after performing a remapping based on a data reordering function. The diamond with an ‘S’ represents a cache hit due to spatial locality. 6
1.5 The data mapping and associated cache behavior after reordering the iterations. The diamond with an ‘T’ represents a cache hit due to temporal locality. .............................. 8
1.6 The data mapping and associated cache behavior after performing a data and iteration reordering ........................................ 8
1.7 Inspector and executor for data reordering example .................. 9
1.8 Compressed sparse row (CSR) matrix format ........................ 12
1.9 Gauss-Seidel for compressed sparse row (CSR) ................. 13
2.1 Example irregular Gauss-Seidel iteration space graph for 3 convergence iterations ........................................ 21
2.2 Example irregular Jacobi iteration space graph for 3 convergence iterations .................................................. 22
2.3 Full sparse-tiled Gauss-Seidel iteration space ...................... 24
2.4 Cache-blocked Gauss-Seidel iteration space ......................... 25
2.5 Typical Gauss-Seidel for CSR assuming an initial reordering of sparse matrix rows and columns and entries in vectors u and f 26
2.6 Code that performs a serial execution of sparse-tiled Gauss-Seidel for compressed sparse row (CSR) ..................... 27
2.7 Compressed sparse row (CSR) matrix format ................. 27
2.8 FULLSPARSENAIVE_GSCSR Algorithm .......................... 41
2.9 FULLSPARSENAIVE_GSCSR with post-conditions, Part I 43
2.10 FULLSPARSENAIVE_GSCSR with post-conditions, Part II .... 44
2.11 FULLSPARSENAIVE_GSCSR with post-conditions, Part III .... 45
2.12 FULLSPARSEWORKSET_GSCSR Algorithm - Part I .... 57
2.13 FULLSPARSEWORKSET_GSCSR Algorithm - Part II .... 58
2.14 MAKESYMMETRICGRAPH Algorithm, which is used in FULLSPARSEWORKSET_GSCSR ................. 59
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.15</td>
<td><strong>UPDATE</strong>θ <strong>ALGORITHM</strong>, which is used in <strong>FULLSPARSE</strong> <strong>WORK</strong>-<strong>SET</strong> <em>GSCSR</em></td>
<td>59</td>
</tr>
<tr>
<td>2.16</td>
<td>On the Power3, normalized execution times of executor as the target seed partition size varies.</td>
<td>67</td>
</tr>
<tr>
<td>2.17</td>
<td>On the UltraSparc-IIi, normalized execution times of executor as the target seed partition size varies.</td>
<td>68</td>
</tr>
<tr>
<td>2.18</td>
<td>On Pentium 4 (2GHz), normalized execution times of executor as the target seed partition size varies.</td>
<td>69</td>
</tr>
<tr>
<td>2.19</td>
<td>On MIPS R10000, normalized execution times of executor as the target seed partition size varies.</td>
<td>70</td>
</tr>
<tr>
<td>2.20</td>
<td>On the Power3, normalized execution times of executor as the number of convergence iterations for the computation and tile growth varies.</td>
<td>71</td>
</tr>
<tr>
<td>2.21</td>
<td>On the UltraSparc-IIi, normalized execution times of executor as the target seed partition size varies.</td>
<td>72</td>
</tr>
<tr>
<td>2.22</td>
<td>On the Pentium 4 (2GHz), normalized execution times of executor as the target seed partition size varies.</td>
<td>73</td>
</tr>
<tr>
<td>2.23</td>
<td>On the MIPS R10000, normalized execution times of executor as the target seed partition size varies.</td>
<td>74</td>
</tr>
<tr>
<td>2.24</td>
<td>Shows how cache-blocked and full sparse-tiled Gauss-Seidel affect the number of TLB, L2 cache, and L1 cache misses on the MIPS R10000.</td>
<td>76</td>
</tr>
<tr>
<td>2.25</td>
<td><strong>FULLSPARSE</strong>_ <strong>NAIVE</strong> _<strong>JACOBI</strong> _<strong>CSR</strong> _ <strong>ALGORITHM</strong></td>
<td>79</td>
</tr>
<tr>
<td>3.1</td>
<td>Two applications of sparse tiling to a two-dimensional iteration space. In both cases, the first row of iterations is partitioned into six cells and then tiles are grown upward. In the top diagram, the seed partitions are ordered from left to right. The small arrows indicate dependences between tiles.</td>
<td>87</td>
</tr>
<tr>
<td>3.2</td>
<td>Tile dependence graphs for the sparse tilings shown in Figure 3.1. 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.</td>
<td>87</td>
</tr>
<tr>
<td>3.3</td>
<td>The average parallelism in the tile dependence graph for full sparse-tiled Gauss-Seidel with 2 convergence iterations. The target seed partition size is 1/8 of L2 cache.</td>
<td>88</td>
</tr>
<tr>
<td>3.4</td>
<td>The effect that graph coloring has on speedup for the parallel full sparse-tiled Gauss-Seidel executor performing 2 convergence iterations. These speedups are from experiments run on one shared memory node of the Blue Horizon at SDSC. Each node has 8 Power3 processors.</td>
<td>89</td>
</tr>
</tbody>
</table>
4.9 Comparing the normalized TLB, L2, and L1 cache misses of sparse-tiled symmetric multigrid with regular symmetric multigrid. The forward and reverse Gauss-Seidel smoothers do 4 convergence iterations at each multigrid level. Both transformations consistently reduce the number of L2 cache misses.

4.10 Normalized execution time of sparse-tiled symmetric multigrid, with sparse tiling overhead included, compared with regular symmetric multigrid. Both the forward and reverse Gauss-Seidel smoothers do 2 convergence iterations at each multigrid level.

4.11 Normalized execution time of sparse-tiled symmetric multigrid, with the sparse-tiling inspector overhead included, compared with regular symmetric multigrid. The forward and reverse Gauss-Seidel smoothers do 4 convergence iterations at each multigrid level.

4.12 Execution time in seconds for the setup and solve of GMRES preconditioned with PETSc SOR and STPetc SOR on the Wing matrix.

4.13 Execution time per GMRES iteration on the Wing Matrix.

4.14 Residual norm after a set number of GMRES iterations (see Table 4.7) for the Wing matrix.

4.15 Execution time in seconds for the setup and solve of GMRES preconditioned with PETSc SOR and STPetc SOR on the 2D Bar matrix.

4.16 Execution time per GMRES iteration on the 2D Bar Matrix.

4.17 Residual norm after a set number of GMRES iterations (see Table 4.7) for the 2D Bar matrix.

4.18 Execution time in seconds for the setup and solve of GMRES preconditioned with PETSc SOR and STPetc SOR on the Sphere matrix.

4.19 Execution time per GMRES iteration on the Sphere Matrix.

4.20 Residual norm after a set number of GMRES iterations (see Table 4.7) for the Sphere matrix.

5.1 Simplified moldyn example.

5.2 Example mapping of iterations in the j loop to locations in the data arrays x and f x. Here, circles represent iterations of the j loop inside one iteration of the outer time-stepping loop.

5.3 Example of Figure 5.2 mapping after the CPACK data reordering.

5.4 Example of Figure 5.2 mapping after the CPACK data reordering followed by a lexGroup iteration reordering.
We highlight the iterations of one sparse tile for the code in Figure 5.1. The j loop has been blocked to provide a seed partitioning. In the full sparse-tiled executor code, the iterations within a tile are executed atomically.

Sparse-tiled executor when the composed inspector performs CPACK, lexGroup, CPACK, lexGroup, full sparse tiling, and tilePack.

Tile packing reorders the iterations and corresponding data locations for the i and k loops so that the computation in each tile has better spatial locality.

Normalized execution time of executor without inspector overhead on the Power3.

Normalized execution time of executor without inspector overhead on the Pentium 4 (1.7GHz).

Amortization of the inspector overhead on the Power3 in number of outer loop iterations based on the savings per iteration experienced by the outer loop in the executor.

Amortization of the inspector overhead on the Pentium 4 1.7GHz in the number of outer loop iterations based on the savings per iteration experienced by the the outer loop in the executor.

First CPACK inspector for moldyn called from composed inspector in Figure 5.13.

Composed inspector for CPACK, lexGroup, CPACK, and lex-Group composition.

Second CPACK inspector for moldyn called from composed inspector in Figure 5.13.

Executor for simple moldyn example when inspector applies CPACK, lexGroup, CPACK, lexGroup composition.

Composed inspector for CPACK, lexGroup, CPACK, and lex-Group where data remapping and index array updates are done immediately after the relevant reordering function is generated.

Percent reduction in inspector overhead for compositions with two or more data reorderings when data is remapped once. nbf does not benefit from the tilePack data reordering transformation; therefore, most of the compositions involving nbf do not use two or more data reordering transformations.
# LIST OF TABLES

1.1 Cache simulation for example index and data arrays ...... 5
1.2 Cache simulation after data reordering .......................... 7
2.1 Data dependence relations for Gauss-Seidel CSR - Part I .... 30
2.2 Data dependence relations for Gauss-Seidel CSR - Part II ... 31
2.3 Data dependence relations for Gauss-Seidel CSR Part III .... 32
2.4 Summary of input matrices used in experiments for sparse tiling stationary iterative methods ................................ 63
2.5 Characteristics of Finite Element Analysis problems .......... 64
2.6 Descriptions of computer architectures used in the serial experiments ............................................... 65
2.7 Descriptions of input matrices used in serial experiments ... 66
2.8 Comparing the best normalized execution times of cache-blocked Gauss-Seidel and full sparse-tiled Gauss-Seidel across various matrices. Any value less than one indicates a reduction in the execution time. The baseline is Gauss-Seidel for CSR with two convergence iterations. The missing results are due to memory limitations on a UltraSparc-III and a Pentium 4 (1.7GHz). ..... 75
2.9 Gives the overhead due to the inspector (OH) and savings incurred by the executor in seconds on the Pentium 4 (2GHz) for two iterations of Gauss-Seidel. The overhead divided by the savings and rounded up is the break even (BE) point, or the number of Gauss-Seidel calls needed to amortize the overhead. We give the results for five different tile growth algorithms. The first three, our implementation of Cache Blocking, FullSparseNaive, and FullSparseWorkSet, only assume that the input sparse matrix is square. The last two, Cache Blocking (Sym) and FullSparseWorkSet (Sym), assume that the sparse matrix is symmetric. .................................................. 77
2.10 Shows the new overhead(OH) and savings results when the original mesh is sparse tiled versus the sparse matrix resulting from FEA. The overhead divided by the savings and rounded up is the break even (BE) point, or the number of Gauss-Seidel calls needed to amortize the overhead. Since the Sphere matrix was not generated by the FETk package, we did not recreate its original mesh. .................................................. 78
3.1 Descriptions of computer architectures used in parallel experiments ............................................. 93
3.2 Descriptions of input matrices used in parallel experiments . . . 93
3.3 The number of Gauss-Seidel executions required to amortize
sparse tiling inspector overhead. Each Gauss-Seidel execution
performs two convergence iterations. ................................. 94
3.4 Percentage of the inspector due to generating the seed parti-
tioning, remapping the data, and growing tiles. Around 80%
to 90% of the overhead is due to partitioning the matrix graph
plus remapping the unknown vector, right-hand side vector, and
sparse matrix. .......................................................... 97
3.5 Summary of how the various Gauss-Seidel parallelization trans-
formations compare in how they handle the three performance
aspects ................................................................. 98
4.1 Descriptions of input matrices used in multigrid experiments . 117
4.2 Descriptions of computer architecture used in symmetric multi-
grid experiments ...................................................... 118
4.3 Summary of the effect of remapping the matrix or not. We
implement the second choice where the matrix is remapped. . 126
4.4 Summary of the effect of where the sparse tiling inspector is
located ............................................................... 128
4.5 Descriptions of computer architectures used in STPetsc experi-
ments ................................................................. 132
4.6 Descriptions of input matrices used in STPetsc experiments . 132
4.7 The number of GMRES iterations used for GMRES precon-
ditioned with the original PETSc SOR implementation to con-
verge with the specified preconditioned relative tolerance (rtol). 132
5.1 Data sets for experiments done with irreg, nbf, and moldyn
benchmarks. foil and auto are used with irreg. moll1 and moll2
are used with nbf and moldyn. ................................. 152
5.2 Descriptions of computer architectures used in composed trans-
formation experiments .............................................. 153
5.3 The baseline execution times in seconds for each processor, pro-
gram, and dataset combination. Below each dataset is the size
of the dataset in the particular program. .......................... 153
Acknowledgments

I have been working with my advisors Jeanne Ferrante and Larry Carter since 1994. The experience has been fun, challenging, enriching, and educational. Jeanne and Larry introduced me to Chai Lattes, taught me how to give talks, helped me improve my writing skills, and introduced me to the exciting world of Computer Science research. Not having a weekly meeting with Jeanne and Larry just won’t be the same. All of the Computer Science and Engineering faculty at UCSD have been supportive and for that I am truly appreciative.

I would also like to thank all my fellow graduate students that helped me throughout the years with their friendship, support, and technical expertise: Nick Mitchell, Barbara Kreaseck, Beth Simon, Sara Miner Moore, Holly Dail, Nadya Williams, Shava Smallen, Michael McCracken, Xiaofeng Gao, Ryan Szypowski, and Sagnik Nandy. The list should be longer, but the dissertation is long enough!

My family has been wonderfully supportive. My parents, Donald and Sharon Mills, have always inspired me to be the best I can possibly be. My many siblings are always there when I need them, especially Cathie and Matt.

My husband Joe was the first person to suggest that I should get a Ph.D. He has supported me so wonderfully that words cannot express my appreciation. With Joe by my side, I can do anything.

I would like to thank Professor Mike Holst for his assistance with the FEtk software package and general information about Finite Element Analysis (FEA). I would like to thank Hwansoo Han for making benchmarks and runtime inspector code available, which I adapted for the experimental results in Chapter 5.

This work was supported by an NSF Graduate Fellowship, an AT&T Labs Graduate Research Fellowship, a grant from Lawrence Livermore National Labs (LLNL), 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). Also, we used Rational PurifyPlus as part of the SEED program.

Chapter 2 is in part a reprint of the material as it appears in the proceedings of the 2001 International Conference on Computational Science (ICCS) and a technical report, and as it will appear in the International Journal of High Performance Computing Applications. The dissertation author was the primary researcher and author for these publications [119, 121, 123], which form the basis for Chapter 2.

Chapter 3 is in part a reprint of the material as it will appear in the proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC). The dissertation author was the primary researcher and author for this publication [122], which forms the basis for Chapter 3.

Chapter 5 is in part a reprint of the material as it appears in the proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). The dissertation author was the primary researcher and author, and the co-authors listed on this publication [120] supervised the research that forms the basis for Chapter 5.
VITA

May 28, 1974  Born
Jackson County, Arkansas

1992  High School Diploma, Bryan High School
Bryan, OH

1994-1996  Senior Coder, University of California, San Diego
La Jolla, CA

1997  B.S. Computer Science
University of California, San Diego
La Jolla, CA

1997  Summer Intern, AT&T Labs Research
Florham Park, NJ

1995, 1998, 1999  Teaching Assistant
University of California, San Diego
La Jolla, CA

2000  M.S. Computer Science
University of California, San Diego
La Jolla, CA

2000  Volunteer High School Instructor
Web Programming
University City High School
San Diego, CA

2001  Summer Instructor
Computer Systems and Organization
University of California, San Diego
La Jolla, CA

2003  Doctor of Philosophy
University of California, San Diego
La Jolla, CA

PUBLICATIONS


Field of Study: Computer Science
ABSTRACT OF THE DISSERTATION
Performance Transformations for Irregular Applications
by
Michelle Mills Strout
Doctor of Philosophy in Computer Science and Engineering
University of California, San Diego, 2003
Professor Larry Carter, Co-Chair
Professor Jeanne Ferrante, Co-Chair

Irregular computations occur in several important science and engineering applications: molecular dynamics simulations, n-body problems, Finite Element Analysis, etc. The observed performance in such irregular applications is typically cited as 10% or less of the advertised peak performance of current computer architectures. These applications frequently use compact sparse matrix formats that inhibit compile-time transformation opportunities. Sparse matrix formats inhibit the static analysis required by compile-time transformations, because their use introduces non-affine memory references such as $A[B[i]]$. Such memory references result in memory access patterns that cannot be determined at compile-time. Therefore, run-time data and computation reordering transformations are needed to improve data locality and exploit parallelism, both of which are essential for improved program performance on current computer architectures.

Run-time reordering transformations are implemented with inspectors and executors. The inspector traverses the memory reference pattern at run-time, generates data and computation reordering functions based on the observed pattern, creates new schedules, and remaps affected data structures accordingly. The executor is a transformed version of the original program that uses the schedules and remapped data structures generated by the inspector. The challenges inherent in using run-time reordering transformations are the need to amortize the
overhead of the inspector, and the automatic composition and generation of the inspector/executor code.

This dissertation makes three main contributions: the development of a run-time reordering transformation for data locality and parallelism called full sparse tiling; a description of technical and software engineering issues that occur when incorporating run-time reordering transformations such as full sparse tiling into existing software packages; and a framework for composing full sparse tiling and other run-time reordering transformations at compile-time. Our results show that on scientific computing benchmarks, compositions of run-time reordering transformations can result in significant performance improvements in serial and parallel processing environments, and that it is possible to amortize the overhead of the inspector.
Chapter 1

Introduction

Many important science and engineering applications exhibit memory reference patterns that are unknown at compile-time. For example, in molecular dynamics simulations each molecule interacts with the subset of molecules close enough in simulation space to have an effect on the molecule’s motion. A list of interacting molecules is calculated based on program input, and then used for some number of simulation time steps before being recalculated. Each simulation time step involves accessing the list of interacting molecules to determine how molecules will change their positions. The list of interactions directly affects the memory access pattern that occurs on the array of molecules, therefore the memory reference pattern cannot be statically determined. Such applications are described as irregular [102].

The problem with irregular applications is that they exhibit poor performance. In fact, only 5-10% of the advertised peak machine performance is typically expected from whole irregular applications [44]. Efforts to improve application performance involve studying kernels within the application. Such kernels have a higher number of floating point operations and therefore might achieve a higher percentage of peak MFLOPS than the 5-10% seen in the whole application. For instance, Mahinthakumar and Saied [111] found that the irreg-
ular bi-conjugate gradient stable solver, which dominates the execution time in their groundwater transport code, exhibits a large drop in performance (from 28% of peak to 14% of peak) when the problem size does not fit into the L2 cache of an SGI Origin 2000. This suggests that one of the causes for the poor performance of irregular applications is poor data locality. Data locality occurs when memory locations mapped to the same cache-line are reused before the cache-line is evicted. The difference in access-time between the cache and memory is one to two orders of magnitude [54, 96, 19, 26], therefore, data locality impacts performance. The study on the bi-conjugate gradient stable solver and other studies have concluded that poor data locality and heavy communication demands contribute toward the performance problem in irregular applications [64, 95].

Compile-time transformations can improve the data locality and exploit parallelism within a program by reordering data and reordering loop iterations [133, 94, 9, 85]. The challenge with irregular programs is that the data and loop iterations can not be effectively reordered at compile time, since the memory access patterns are unknown until runtime. Therefore, run-time reordering transformations have been developed to improve the data locality and exploit the parallelism within irregular programs. Such transformations are implemented with inspector/executor strategies [90]. The inspector traverses the memory reference pattern at runtime, generates data-reordering and iteration-reordering functions based on the observed pattern, creates new schedules, and remaps affected data structures accordingly. The executor is a transformed version of the original program that uses the schedules and remapped data structures generated by the inspector.

Ideally the compiler would automatically analyze irregular applications to determine the applicability of run-time reordering transformations, generate the inspector and executor code to implement any applicable composition of
transformations, and generate decision code that will determine at runtime what compositions of such transformations (if any) will improve the overall application performance (see Figure 1.1). Current research in this area describes the analysis and code generation algorithms for *specific* run-time reordering transformations and *specific* compositions of such transformations.

In this dissertation, we make three main contributions toward the challenge of improving irregular application performance. First, we develop the *full sparse tiling* (FST) run-time reordering transformation for use with stationary iterative solvers like Gauss-Seidel. Full sparse tiling improves the Gauss-Seidel executor performance by improving data locality and exploiting parallelism. Second, we implemented sparse tiling in two existing software packages: the Finite Element toolkit (FEtk) [60] and the Portable, Extensible Toolkit for Scientific Computation (PETSc) [11]. By using FEtk to generate hierarchies of sparse matrices, we explore the performance benefits and trade-offs involved in using sparse-tiled Gauss-Seidel within symmetric multigrid. We also implement a sparse tiling extension to PETSc to explore the software engineering issues involved in incor-
for $i = 0, 7$ do
    $Y[i] = Z[x[i]]$
endo

Figure 1.2: Simple irregular loop with one indirect memory reference

Incorporating run-time reordering transformations into existing software packages. Finally, we develop a framework for expressing and composing run-time reordering transformations at compile time. The framework is an important step toward creating a compiler that can automatically generate inspectors and executors for legal compositions of run-time reordering transformations.

1.1 Background

Run-time reordering transformations have been successfully used to improve data locality [7, 33, 91, 35, 50, 119, 65] and exploit parallelism [31, 103, 52, 47]. In this section, we review the concepts of run-time reordering transformations with an example. Figure 1.2 exhibits a simple irregular loop with an indirect memory reference, the data array $Z$ referenced by the index array $x$. The loop is irregular because the memory access pattern of the $Z$ data array is dependent on the values in the index array $x$, which in turn depend on program input data. Compile-time reordering transformations such as traditional tiling [132, 66, 41, 131, 129, 21, 86] are not applicable to irregular loops because such transformations require compile-time data dependence analysis.

In almost all modern computers, whenever a memory location is referenced by a program, the data in the referenced location and nearby locations are brought into a fast but small data cache.\footnote{In fact, most computers have several caches, including a very fast L1 cache that can hold tens of kilobytes of data, and a slightly slower L2 cache that can hold around a megabyte.} Any additional references to data already in the cache-line (before the cache-line is evicted from the cache) will be
Figure 1.3: The data mapping and associated cache behavior for the original computation. The circles represent iterations in the loop and the M's represent cache misses due to the memory reference into the data array \(Z\).

<table>
<thead>
<tr>
<th>(i)</th>
<th>Hit or Miss</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Miss</td>
<td>c d</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>c d g h</td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>a b g h</td>
</tr>
<tr>
<td>3</td>
<td>Miss</td>
<td>a b e f</td>
</tr>
<tr>
<td>4</td>
<td>Miss</td>
<td>c d e f</td>
</tr>
<tr>
<td>5</td>
<td>Miss</td>
<td>c d g h</td>
</tr>
<tr>
<td>6</td>
<td>Miss</td>
<td>a b g h</td>
</tr>
<tr>
<td>7</td>
<td>Miss</td>
<td>a b c d</td>
</tr>
</tbody>
</table>

Table 1.1: Cache simulation for example index and data arrays

one or two orders of magnitude faster then references to main (DRAM) memory. When such usage occurs during the execution of a program it is referred to as *data locality*.

The effect of example values in the index array on cache performance can be seen through the use of a simplified cache simulation. Assume that each cache-line holds two elements of the data array \(Z\), and that there is room for two cache-lines in the cache at any one time with least recently used (LRU) cache-line
Figure 1.4: The data mapping and associated cache behavior after performing a remapping based on a data reordering function. The diamond with an ‘S’ represents a cache hit due to spatial locality.

replacement 2. Figure 1.3 shows that every iteration of the simple loop results in a cache miss (M) for the example index array x. The details of the simulation, in terms of what data is located in the cache after each loop iteration, are shown in Table 1.1.

Intuitively, a program has good data locality if most of the time, the data referenced by the program is in its cache. The locality of a program can be improved by changing the order of computation (iteration reordering) and/or the assignment of data to memory locations (data reordering) so that references to the same or nearby locations occur relatively close in time during the execution of the program.

In general, run-time data-reordering transformations improve the spatial locality in a computation. Spatial locality occurs when memory locations mapped to the same cache-line are used before the cache-line is evicted. Figure 1.4 and Table 1.2 show the results of the simple cache simulation after the data array Z has been remapped based on a data reordering function generated with the con-

---

2For simplicity, we ignore the cache activity needed to access the other data items such as the index array x and the data array Y).
secutive packing algorithm [33]. The diamond with an ‘S’ in Figure 1.4 indicates a cache hit due to spatial locality. The consecutive packing algorithm creates a data reordering function that packs data items into the remapped data array based on the order the data items will be accessed by the loop iterations. After applying a data reordering transformation, the data items are still accessed in the same order (d, h, a, f, ...), but now the accesses experience more spatial locality and therefore fewer cache misses.

Temporal locality occurs when the same memory location is reused before its cache-line is evicted. In general, run-time iteration reordering can improve the temporal and spatial locality within an irregular application. Figure 1.5 shows the results of the simple cache simulation after the iterations in the i loop have been reordered. The diamond containing the letter ‘T’ represents a cache hit due to temporal locality. The iteration reordering algorithm in this example is lexicographical grouping [33], which groups iterations based on the data locations they access. In this example where there is only one indirect memory reference, the index array x is sorted by value. It is also possible to combine data and iteration reordering transformations. Figure 1.6 shows the result of the simple cache simulation after a consecutive packing data reordering is followed by a lexicographical grouping iteration reordering.
Figure 1.5: The data mapping and associated cache behavior after reordering the iterations. The diamond with an ‘T’ represents a cache hit due to temporal locality.

Figure 1.6: The data mapping and associated cache behavior after performing a data and iteration reordering
**INSPECTOR**

for $i = 0, 7$ do  
... $x[i]$ ...  
for $j = 0, 7$ do  
$\sigma[j] = ...$  
endfor  

for $j = 0, 7$ do  
$Z'[\sigma[j]] = Z[j]$  
$x'[j] = \sigma[x[j]]$  
endfor  

**EXECUTOR**

for $i = 0, 7$ do  
$Y[i] = Z'[x'[i]]$  
enddo

---

Figure 1.7: Inspector and executor for data reordering example

Figure 1.7 exhibits the inspector and executor for the data-reordering transformation example in Figure 1.4. The inspector traverses the index array $x$ and using a data-reordering algorithm such as consecutive packing [33] generates a data-reordering function $\sigma$, which is stored in the array $\sigma$. The inspector is also responsible for remapping the data arrays according to the data reordering function and updating the index array to refer to the new data locations. The executor is the original code rewritten so that it uses the new index and data arrays created by the inspector.

The overhead of run-time reordering transformations is the execution time of the inspector. For run-time reordering transformations to exhibit an overall reduction in execution time, the remapped data and/or new iteration schedule must be reused enough times that the overhead of the inspector is amortized. This is referred to as schedule reuse [31]. In science and engineering applications, irregular memory references often occur in nested loops and therefore schedule
reuse is possible across the outer loops.

Parallelism can also be exploited. For the example in Figure 1.2, shared memory parallelism is possible and simple to implement since there are no data dependences between iterations of the loop. Because the irregular references occur in a memory read, compile-time analysis would also detect the possibility of parallelizing this simple loop. For distributed memory parallelism, it is necessary to assign data and iterations to processors, and then inspectors and executors can be used to determine the required communication pattern and perform the requisite communication. Distributed memory parallelism is not within the scope of this dissertation.

Inspector/executor strategies for run-time reordering transformations were first developed to parallelize irregular applications [90, 112]. Steps toward automating the application of such transformations include using libraries of run-time reordering transformation primitives [29, 63, 62], and automatic insertion of calls to such libraries [31]. More recently run-time reordering transformations have been developed for improving the data locality within general programs [91, 33, 50] and the Gauss-Seidel computation [35, 119]. Such work along with more recent work in run-time reordering for parallelism [102, 52] has the compiler analyze the code and generate the inspector/executor code for specific run-time reordering transformations. The framework for compile-time composition of run-time reordering transformations, presented in this dissertation, will allow for the next level of automation.

1.2 Sparse Tiling for Stationary Iterative Methods

One of the contributions of this dissertation is the development of the run-time reordering transformation full sparse tiling. Full sparse tiling improves the data locality and consequently the performance of stationary iterative meth-
ods operating on sparse matrices. In many scientific applications, such as Finite Element Analysis, these iterative methods dominate the execution time. The computation done within stationary iterative methods is irregular because it involves traversing a sparse matrix data structure.

Stationary iterative methods such as Gauss-Seidel, Successive Overrelaxation (SOR), and Jacobi can be used to solve systems of simultaneous linear equations [14]. A system of linear equations is represented as $A\vec{u} = \vec{f}$, where $\vec{u}$ is a vector of unknowns, $A$ is a sparse matrix of coefficients, and $\vec{f}$ is a vector of known constants referred to as the right-hand side. Stationary iterative methods solve for the unknown vector $\vec{u}$ by iteratively applying an approximate inverse for the matrix $A$, converging toward a solution. The algorithm for Gauss-Seidel is shown in Equation (1.1), with $u_v$ representing an element in the unknown vector $\vec{u}$, $f_v$ an element in the right-hand side vector $\vec{f}$, and $a_{vw}$ the element at row $v$ and column $w$ in the square matrix $A$. We refer to each of the $T$ iterations of the outermost loop as a convergence iteration. The $v$ loop iterates over the $R$ unknowns and corresponding rows in matrix $A$. The $w$ loop, which is implicit in the summations, iterates over the $R$ columns of matrix $A$. Note that the first summation uses values of $u_w$ that were computed earlier in the current convergence iteration, while the second uses values computed in the previous convergence iteration. Upon the completion of each convergence iteration a new value has been generated for all the unknowns in $\vec{u}$.

$$\text{for } \text{iter} = 1, 2, ..., T$$
$$\text{for } v = 0, 1, ..., (R - 1)$$
$$u_v = (1/a_{vv}) \left( f_v - \sum_{w=0}^{v-1} a_{vw}u_w - \sum_{w=v+1}^{R-1} a_{vw}u_w \right) \quad (1.1)$$

In many applications involving sparse matrices, fewer than 1% of the possible entries of the matrix are nonzeros [99, 64]. A sparse matrix format
Figure 1.8: Compressed sparse row (CSR) matrix format

saves computation time and storage, because only the nonzero entries of the
matrix are stored and traversed within the computation. There are many sparse
matrix formats [106], but this dissertation focuses on square matrices stored in
compressed sparse row (CSR), which stores the nonzeros of the matrix by row
(see Figure 1.8). Figure 1.9, which is discussed further in Section 2.3, shows how
Gauss-Seidel can be implemented using CSR. Although sparse matrix formats
save a large amount of computation time and storage, their use in an application
results in memory reference patterns that cannot be analyzed statically. One
such reference, \( u[ja[p]] \), can be seen at Statement 2 in Figure 1.9.

For improved performance in stationary iterative methods, run-time re-
ordering transformations can be used to exploit three different performance as-
pects: intra-iteration data locality, inter-iteration data locality, and parallelism.
In the implementation of Gauss-Seidel of Figure 1.9, there are opportunities to
improve the data locality within a convergence iteration, *intra-iteration locality*,
and between convergence iterations, *inter-iteration locality*. The order in which
elements of the unknown vector are visited and stored affects intra-iteration lo-
cality. The typical schedule for Gauss-Seidel, as shown in Figure 1.9, completes
**GaussSeidelCSR**($A(\text{ia,ja,a}),\text{u,f}$)

```plaintext
for iter = 1, T do
   for i = 0, (R − 1) do
      \( u[i] = f[i] \)
      for \( p=\text{ia}[i], \text{ia}[i + 1]−1 \) do
         if \( ( \text{ja}[p] \neq i ) \) then
            \( u[i] = u[i] - a[p] \times u[\text{ja}[p]] \)
         else
            \( \text{diag}[i] = a[p] \)
         endif
      endfor
   endfor
   \( u[i] = u[i]/\text{diag}[i] \)
endfor
```

Figure 1.9: Gauss-Seidel for compressed sparse row (CSR)

all the computation for one convergence iteration before doing any computation for the next. Because the unknown vector \( \vec{u} \), right-hand side vector \( \vec{f} \), and sparse matrix \( A \) are often quite large, it is very unlikely that much inter-iteration locality occurs in a typical Gauss-Seidel implementation. Stationary iterative methods like Gauss-Seidel contain parallelism inherent in the fact that the computations for an individual unknown only depend on a subset of the computations for other unknowns. Section 2.1 describes how to visualize the computation for stationary iterative smoothers and points out the parallelization opportunities.

Full sparse tiling [119], developed as part of this dissertation, and cache blocking for irregular grids [35], developed by Douglas et al., are run-time reordering transformations developed for Gauss-Seidel and applicable to all stationary iterative methods. We refer to both these transformations as sparse tiling transformations because their executor has been transformed from the original code to one that executes the iteration space in a tile-by-tile fashion. Each tile contains computations from multiple convergence iterations for a subset of the unknowns.
Since sparse matrix formats are involved, the actual mapping of iteration points to tiles is performed at runtime. The main difference between full sparse tiling and cache blocking is how the tiles are grown throughout the iteration space. By selecting the sparse tiles for improved inter-iteration locality, reordering the unknown vector and other data structures for intra-iteration locality, and executing the independent tiles in parallel, all three performance aspects can be obtained with sparse tiling transformations.

It is necessary to reorder the unknowns to apply sparse tiling transformations. A data dependence occurs between two computations if they access the same memory location and one of the accesses is a write. In the case of Gauss-Seidel, reordering the unknowns actually affects the data dependences within the computation. Specifically, reordering the unknowns will result in each unknown depending on the same set of unknowns, but whether they will depend on the computation from the previous convergence iteration or the current convergence iteration could change. Also, since the columns are reordered, the order of the summation will change as well. The ability to a priori reorder the unknowns in Gauss-Seidel is an example of domain-specific knowledge being used for program transformations. Specifically, the order of the unknowns is in most cases arbitrary, and therefore, a reordering of the unknowns will result in similar convergence properties [135].

1.3 Using Sparse Tiling in Existing Software Packages

Gauss-Seidel and other stationary iterative methods are used as subroutines by other algorithms. Multigrid methods and Krylov methods are two sets of algorithms that utilize stationary iterative methods. In Chapter 4 we explore the performance benefits and tradeoffs of implementing sparse-tiled Gauss-Seidel as the smoother for a symmetric multigrid implementation within the Finite Ele-
ment toolkit (FETk) [60] software package. We also describe some of the software engineering issues that occur when implementing sparse-tiled SOR as an extension to the Portable, Extensible Toolkit for Scientific Computation (PETSc) [11] software package.

In the FETk software package, multigrid methods are used within the context of Finite Element Analysis (FEA). FEA is used for solving partial differential equations (PDEs). By discretizing the PDEs over a domain consisting of finite elements such as triangles, a simultaneous system of linear equations is generated.

Multigrid methods are frequently used in FEA to solve the resulting systems of simultaneous linear equations. Multigrid algorithms call Gauss-Seidel multiple times with the same sparse matrix as input, thus it is possible to amortize the overhead of the sparse tiling inspector. Douglas et al. [35] showed that cache blocking helps the overall performance of multigrid, however the inspector overhead studied was that of partitioning and reordering the meshes at each multigrid level. Only geometric multigrid methods generate a mesh for each level of multigrid. When there are multiple unknowns per mesh node, the assembled sparse matrix is considerably larger (approximately $d^2$ larger where $d$ is the number of unknowns per mesh node) than the mesh. In Section 4.1, we consider whether sparse tiling transformations can realistically amortize the overhead required with the assumption that only the sparse matrices are available at each multigrid level. In this context, amortizing the overhead of sparse tiling transformations is more difficult, and as our results show it isn’t always possible in one call to multigrid.

There is a variant of multigrid, called symmetric multigrid, that is frequently used as a preconditioner for Conjugate Gradient [2]. Symmetric multigrid uses a forward and reverse ordering of the Gauss-Seidel computation. We show
how to use the run-time iteration and data reordering provided by either full sparse tiling or cache blocking in reverse Gauss-Seidel.

In Section 4.2, we describe the software engineering issues involved in implementing sparse-tiled SOR as an extension to the PETSc software package. The PETSc library already handles many performance issues such as parallelization, sparse matrix formats, usage levels, and others. We explore how other performance issue such as using the full sparse tiling transformation might be incorporated while maintaining the established software architecture. The initial implementation of our extension (STPetsc) includes two interface levels, a level where the user just specifies that sparse tiling should be applied, and another level where decisions involving when the sparse tiling inspector executes have been exposed to the user.

1.4 Run-time Reordering Transformation Framework

Data locality and parallelism are not only important for stationary iterative methods; they are essential for the performance of all irregular applications on current computer architectures. Therefore, we also explore techniques for applying run-time reordering transformations more generally without the use of domain-specific information such as the a priori reordering information used in sparse tiling transformations for Gauss-Seidel. Specifically we develop a compile-time framework for manipulating and composing run-time reordering transformations.

Until recently the focus in the compiler research community has been primarily on compile-time transformation frameworks [83, 113, 80, 22, 73, 77, 130, 124] restricted to affine loop bounds and affine array references. These frameworks allow for the uniform representation, the composition, the legality determination, and sometimes a benefit model of various compile-time transformations.
Many of these frameworks represent loop nests as iteration spaces. Loop transformations are then expressed as mappings between iterations spaces. The legality of a transformation is determined by the transformed data dependences of the program. Compile-time frameworks for compile-time transformations conservatively assume there is a dependence when faced with non-affine memory references. One such framework — that of Kelly and Pugh [73] — describes non-affine memory references (such as indirect memory references $A[B[i]]$) by using Presburger arithmetic with uninterpreted function symbols [97]. We exploit this ability to specify data mappings between loop iterations and data locations, and dependences between loop iterations, when non-affine memory references are involved. Our framework can also express run-time data-reordering and iteration-reordering transformations, such as consecutive packing [33], graph partitioning [50], bucket tiling [91], lexicographical grouping [33], full sparse tiling [119], and cache blocking [35].

Describing the effect of run-time data and iteration reorderings within a compile-time framework has several advantages. First, both run-time and compile-time transformations are uniformly described. Secondly, the transformation legality checks provide constraints on the run-time reordering functions. Finally, the overhead involved in generating the run-time reordering functions can be reduced with various optimizations, such as moving the data to new locations only once and traversing fewer dependences.

Our key insight is that given a composition of run-time reorderings, the modified data mappings and/or dependences are used by inspectors for subsequent run-time reordering transformations. This allows for compile-time composition of run-time reorderings. Our experiments (on the irreg, nbf, and moldyn benchmarks [50]) illustrate that, in some cases, large performance improvements result from composite run-time reordering transformations.
1.5 Contributions

We make the following contributions in this thesis:

- Provide an inspector/executor framework for implementing two sparse tiling transformations: full sparse tiling [119] and cache blocking for irregular grids [35].

- Prove that full sparse-tiled Gauss-Seidel generates bit-equivalent results to traditional Gauss-Seidel with the same a priori ordering of unknowns.

- Give serial and parallel experimental results that indicate full sparse tiling improves the serial performance of Gauss-Seidel on many different machines.

- Experimentally show that sparse-tiled Gauss-Seidel has better speedup than Gauss-Seidel parallelized with an owner-computes method [57].

- Describe how sparse tiling transformations can be used within the context of symmetric multigrid, and present experimental results from our extension to FEtk [60].

- Implement a sparse tiling extension to PETSc [11] to explore the software engineering issues involved in incorporating run-time reordering transformations into existing software packages.

- Give experimental results (using hand-coded inspectors and executors) showing that significant performance improvements can result from composing run-time reordering transformations.

- Show how to use the existing compile-time framework of Kelly and Pugh to describe a number of existing run-time data and iteration-reordering transformations that improve data locality. We also describe the space of possible run-time reordering transformations and how existing transformations fit within it.
• Show how sparse tiling transformations, which improve locality in loops and between loops that carry data dependences, can be described in this framework.

• Give experimental results that show moving the data to new locations only once reduces the overhead of composed run-time reordering transformations.

1.6 Dissertation Overview

Chapter 2 presents the legality and serial efficiency of the run-time reordering transformation full sparse tiling when applied to the stationary iterative method Gauss-Seidel. Chapter 3 extends the use of full sparse tiling for shared-memory parallelism in Gauss-Seidel. In Chapter 4, we explore the performance benefits and tradeoffs involved in using sparse-tiled Gauss-Seidel with symmetric multigrid using the FEtk [60] software package. We then explore the software engineering issues that occur when implementing sparse-tiled SOR as an extension to the PETSc [11] software package. We present a framework for the compile-time composition of run-time reordering transformations including sparse tiling transformations in Chapter 5. In Chapter 6, we conclude and describe possible future work based on the work presented in this dissertation.
Chapter 2

Sparse Tiling Stationary Iterative Methods for Locality

This chapter details full sparse tiling, a run-time reordering transformation that improves data locality and exploits parallelism for stationary iterative methods such as Gauss-Seidel operating on square sparse matrices. In scientific applications such as Finite Element Analysis, these iterative methods dominate the execution time. Full sparse tiling creates a reordering of the rows and columns of the sparse matrix, and also creates an iteration reordering for the execution. We prove that full sparse-tiled Gauss-Seidel generates a solution that is bitwise identical to traditional Gauss-Seidel on the remapped matrix. We also present measurements of the performance improvements and the overheads incurred by two sparse tiling transformations: full sparse tiling and cache blocking for irregular grids developed by Douglas et al.

In Sections 2.1 and 2.2, we describe how to visualize the Gauss-Seidel computation and sparse tilings of Gauss-Seidel respectively. Sections 2.3 and 2.4 overview the inspector/executor framework for implementing both sparse tiling transformations and present the proof of correctness for full sparse tiling. In
Section 2.6 we describe experiments on a variety of machines that investigate the issues involved in selecting parameters for both sparse tiling transformations, a comparison of the performance improvements observed in the executors, and the overhead of the inspectors. Finally, we describe how full sparse tiling can be applied to other stationary iterative methods in Section 2.7, present related work in Section 2.8, and summarize in Section 2.9.

2.1 Visualizing Stationary Iterative Methods

To illustrate sparse tiling transformations, we first visualize the Gauss-Seidel computation as specified in Equation (2.1).
Figure 2.2: Example irregular Jacobi iteration space graph for 3 convergence iterations

\[ u_v = \left( 1/a_{vv} \right) \left( f_v - \sum_{w=0}^{v-1} a_{vw}u_w - \sum_{w=v+1}^{R-1} a_{vw}u_w \right) \] (2.1)

The nonzero structure of a sparse matrix represents the relationships between the unknowns in vector \( \vec{u} \). In the corresponding matrix graph \( G(V, E) \), there is a node \( v \in V \) for each unknown \( u_v \) and an edge \( <v, w> \in E \) for each nonzero in the sparse matrix \( A \). The Gauss-Seidel computation on a sample sparse matrix can be visualized with the iteration space graph in Figure 2.1, where each convergence iteration is represented by a copy of the matrix graph. Each iteration point\(^1\), \( <iter, v> \), represents the computations for \( u_v \) at convergence iteration \( iter \) as

\(^1\)We use the term iteration point for points in the iteration space graph and node for points in the matrix graph.
specified in Equation (2.1). A data dependence, represented by an arrow, occurs when one iteration point computes a value that is used in another iteration point. For clarity, only the data dependences for the iteration points associated with one matrix node are shown in Figures 2.1 and 2.2.

In Gauss-Seidel, each computation for \( u_v \) uses the most recently calculated values of the neighboring unknowns; therefore some data dependences come from iteration points in the same convergence iteration, and some come from points in the previous convergence iteration. The ordering of the unknowns affects whether the dependence on a neighboring unknown comes from the last convergence iteration or the previous convergence iteration. In Jacobi (Figure 2.2), each computation for \( u_v \) depends only on the iteration points of its neighbors in the matrix graph from the previous convergence iteration.

### 2.2 Visualizing Sparse Tiling for Gauss-Seidel

There are two run-time reordering transformations we characterize as sparse tiling transformations. **Full sparse tiling** is a run-time reordering transformations that we develop as part of this dissertation. Douglas et al. [35] describe another sparse tiling transformation, which they refer to as cache blocking of unstructured grids. In this dissertation, we refer to their transformation as **cache blocking**. Figures 2.3 and 2.4 illustrate how the full sparse tiling and the cache blocking transformations 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.

To parallelize cache-blocked Gauss-Seidel, the “pyramid”-shaped tiles in Figure 2.4 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 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 transformations within the same framework, therefore, we can compare their effectiveness for serial and parallel executions. Serial experimental results are given in this chapter, and parallel experimental results are given in Chapter 3.

2.3 Sparse Tiling Executor for Gauss-Seidel

Inspector/executor strategies were initially developed for parallelizing applications with irregular memory references [31]. For sparse tiling, the inspector
Figure 2.4: Cache-blocked Gauss-Seidel iteration space

examines the nonzero structure of the sparse matrix $A$ at runtime, generates a data reordering and a schedule based on a tiling function, and remaps the sparse matrix and vectors based on the data reordering. The *executor* is a transformed version of the original code that uses the remapped matrix and vectors and the schedule created by the inspector.

In this section, we describe how Gauss-Seidel for the compressed sparse row (CSR) format is transformed at compile time to generate the sparse tiling executor. Theorem 1 in Section 2.3.3 constrains the tiling function so that a serial tile-by-tile schedule generates an unknown vector that is bit-equivalent to what is generated from a typical Gauss-Seidel schedule that uses the same data ordering. The various algorithms involved in the sparse tiling inspector and the proof that they satisfy the constraints given in Theorem 1 are detailed in Section 2.4.

The original order, $v = 0, 1, ..., (R - 1)$, given to the unknowns and
Figure 2.5: Typical Gauss-Seidel for CSR assuming an initial reordering of sparse matrix rows and columns and entries in vectors $\mathbf{u}$ and $\mathbf{f}$

corresponding matrix rows and columns is often arbitrary and can be changed without affecting the convergence properties of Gauss-Seidel [135]. Therefore, if the unknowns are mapped to another order before performing Gauss-Seidel, the final numerical result will vary somewhat, 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 compare the result of sparse-tiled Gauss-Seidel using the generated reordering, $i = \sigma(v)$ where $i$ is the loop iterator in Figures 2.5 and 2.6, to that of typical Gauss-Seidel using the same reordering. Therefore, in Figures 2.5 and 2.6, assume that the original matrix $A$, unknown vector $\tilde{u}$, and right-hand side $\tilde{f}$ have been remapped 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 2.5 gives detailed pseudocode for typical Gauss-Seidel written for the compressed sparse row (CSR) matrix format. Using CSR the nonzeros in the sparse matrix $A'$ are stored by row using three one-dimensional arrays $\mathbf{ia}, \mathbf{ja},$ and

<table>
<thead>
<tr>
<th>\text{GaussSeidelCSR}(A'(\mathbf{ia,j}\mathbf{a,a}),\mathbf{u}',\mathbf{f}')</th>
<th>\text{for}_{\text{iter}} = 1, T \text{ do}</th>
<th>\text{for } i = 0, (R - 1) \text{ do}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbf{u}'[i] = \mathbf{f}'[i]$</td>
<td>$\mathbf{u}'[i] = \mathbf{u}'[i] - \mathbf{a}[p] * \mathbf{u}'[\mathbf{ja}[p]]$</td>
<td></td>
</tr>
<tr>
<td>\text{for } p=\mathbf{ia}[i], \mathbf{ia}[i+1]-1 \text{ do}</td>
<td>\text{else}</td>
<td></td>
</tr>
<tr>
<td>\text{if } ( \mathbf{ja}[p] \neq i ) \text{ then}</td>
<td>\text{endif}</td>
<td></td>
</tr>
<tr>
<td>$\mathbf{diag}[i] = \mathbf{a}[p]$</td>
<td>$\mathbf{u}'[i] = \mathbf{u}'[i]/\mathbf{diag}[i]$</td>
<td></td>
</tr>
<tr>
<td>\text{endfor}</td>
<td>\text{endfor}</td>
<td></td>
</tr>
</tbody>
</table>

```
GaussSeidelCSR_ST(A'(ia,ja,a),u',f',sched,k)
for tileID = 0, (k - 1) do
  for iter = 1, T do
    for i ∈ sched(tileID, iter) do
      1: u'[i] = f'[i]
         for p = ia[i], ia[i + 1] - 1 do
           if ( ja[p] ≠ 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
endfor

Figure 2.6: Code that performs a serial execution of sparse-tiled Gauss-Seidel for compressed sparse row (CSR)

Figure 2.7: Compressed sparse row (CSR) matrix format
a (see Figure 2.7). The nonzeros of the ith row of A’ are stored in the a array, from locations ia[i] up to ia[i + 1]-1. The corresponding entries of ja indicate the nonzero’s column. In Figure 2.5, the vectors u’ and f’ are represented with the arrays u’ and f’.

At run-time, sparse tiling inspectors generate a data reordering function, \( \sigma(v) : V \rightarrow \{0, ..., (R - 1)\} \) and a tiling function, \( \theta(iter, v) : \{1, ..., T\} \times V \rightarrow \{0, ..., (k - 1)\} \). The tiling function maps iteration points \(<iter, v>\) to tiles. From this tiling function, the inspector creates a schedule function, \( \text{sched}(tileID, iter) : \{0, ..., (k - 1)\} \times \{1, ..., T\} \rightarrow 2^{\{0, ..., (R - 1)\}} \). The schedule function specifies for each tile and convergence iteration the subset of the reordered unknowns that must be updated. The transformed code shown in Figure 2.6 performs a tile-by-tile execution of the iteration points using the schedule function, which is created by the inspector to satisfy the following:

\[
\text{sched}(tileID, iter) = \{\sigma(v) \mid \theta(iter, v) = tileID\}.
\]

To prove the correctness of the sparse-tiled Gauss-Seidel in Figure 2.6, we first determine the symbolic data dependence relations for the Gauss-Seidel pseudocode shown in Figure 2.5. We then specify the compile-time mappings [73] that transform typical Gauss-Seidel for CSR to sparse-tiled Gauss-Seidel for CSR. Finally, we determine the constraints that the run-time generated tiling function \( \theta \), and corresponding schedule function \( \text{sched} \), must satisfy to make the transformation legal. A transformation is legal if the transformed data dependences are satisfied by a lexicographical execution of the transformed iteration space.

2.3.1 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 2.5 lies within the iteration space \( \{[iter, i] \mid (1 \leq iter \leq T) \text{ and } (0 \leq i < R)\} \). Data dependences indicate when an
instance of a statement must execute before the instance of another (possibly the
same) statement to preserve the program’s semantics. In our context, data de-
pendence relations are sets of data dependences between statement instances [73]
expressed as mappings. For example, since Statement 1 in Figure 2.5 writes to
the same memory location at each convergence iterations $iter$, there is a de-
pendence between any instance of Statement 1 to all later instances of Statement 1.
These dependences 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 [97]. 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 $u'[i]$ in
Statement 1 of Figure 2.5 and the read of $u'[ja[p]]$ in Statement 2 is as follows:

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

and $ia(i) \leq p < ia(i + 1) \text{ and } i \neq ja(p)\}.$

Tables 2.1 through 2.3 list all of the data dependence relations for the typical
Gauss-Seidel implementation for CSR in Figure 2.5. We calculated the memory-
based data dependences as described in [100]. We verified their correctness by
comparing them to the data dependence relations generated by Petit [70], which
is a research tool for performing data dependence analysis.

In order to perform Gauss-Seidel on the matrix $A'$ there must be nonzero
values on the diagonals. Therefore, we assume that the write to the array variable
diag in Statement 3 of Figure 2.5 occurs at least once for each iteration of the
$i$ loop. Since $\text{diag}[i]$ is assigned and then used within each $i$ iteration, data
<table>
<thead>
<tr>
<th>ID</th>
<th>Ref 1</th>
<th>Ref 2</th>
<th>Data dependence relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>1: (u'[i])</td>
<td>1: (u'[i])</td>
<td>({[\text{iter}_1, i] \rightarrow [\text{iter}_2, i] \mid 1 \leq \text{iter}_1 &lt; \text{iter}_2 \leq T \text{ and } 0 \leq i &lt; R})</td>
</tr>
<tr>
<td>D2</td>
<td>1: (u'[i])</td>
<td>2: (u'[i]) (read)</td>
<td>({[\text{iter}_1, i] \rightarrow [\text{iter}_2, i, p] \mid 1 \leq \text{iter}_1 \leq \text{iter}_2 \leq T \text{ and } 0 \leq i &lt; R \text{ and } \text{ia}(i) \leq p &lt; \text{ia}(i + 1) \text{ and } i \neq ja(p)})</td>
</tr>
<tr>
<td>D3</td>
<td>1: (u'[i])</td>
<td>2: (u'[i]) (write)</td>
<td>same as dependence relation D2</td>
</tr>
<tr>
<td>D4</td>
<td>1: (u'[i])</td>
<td>2: (u'[ja[p]])</td>
<td>({[\text{iter}, i_1] \rightarrow [\text{iter}, i_2, p] \mid 1 \leq \text{iter} \leq T \text{ and } 0 \leq i_1 &lt; i_2 &lt; R \text{ and } \text{ia}(i_2) \leq p &lt; \text{ia}(i_2 + 1) \text{ and } i_1 = ja(p)} \cup {[\text{iter}_1, i_1] \rightarrow [\text{iter}_2, i_2, p] \mid 1 \leq \text{iter}_1 &lt; \text{iter}_2 \leq T \text{ and } 0 \leq i_1, i_2 &lt; R \text{ and } i_1 \neq i_2 \text{ and } \text{ia}(i_2) \leq p &lt; \text{ia}(i_2 + 1) \text{ and } i_1 = ja(p)})</td>
</tr>
<tr>
<td>D5</td>
<td>1: (u'[i])</td>
<td>4: (u'[i]) (read)</td>
<td>({[\text{iter}_1, i] \rightarrow [\text{iter}_2, i] \mid 1 \leq \text{iter}_1 \leq \text{iter}_2 \leq T \text{ and } 0 \leq i &lt; R})</td>
</tr>
<tr>
<td>D6</td>
<td>1: (u'[i])</td>
<td>4: (u'[i]) (write)</td>
<td>same as dependence relation D5</td>
</tr>
<tr>
<td>D7</td>
<td>2: (u'[i]) (read)</td>
<td>1: (u'[i])</td>
<td>({[\text{iter}_1, i, p] \rightarrow [\text{iter}_2, i] \mid 1 \leq \text{iter}_1 &lt; \text{iter}_2 \leq T \text{ and } 0 \leq i &lt; R \text{ and } \text{ia}(i) \leq p &lt; \text{ia}(i + 1) \text{ and } i \neq ja(p)})</td>
</tr>
<tr>
<td>D8</td>
<td>2: (u'[i]) (read)</td>
<td>2: (u'[i]) (read)</td>
<td>read-read dependence</td>
</tr>
<tr>
<td>D9</td>
<td>2: (u'[i]) (read)</td>
<td>2: (u'[i]) (write)</td>
<td>({[\text{iter}, i, p_1] \rightarrow [\text{iter}, i, p_2] \mid 1 \leq \text{iter} \leq T \text{ and } 0 \leq i &lt; R \text{ and } p_1 \leq p_2 \text{ and } \text{ia}(i) \leq p_1, p_2 &lt; \text{ia}(i + 1) \text{ and } i \neq ja(p_1) \text{ and } i \neq ja(p_2)} \cup {[\text{iter}_1, i, p_1] \rightarrow [\text{iter}_2, i, p_2] \mid 1 \leq \text{iter}_1 &lt; \text{iter}_2 \leq T \text{ and } 0 \leq i &lt; R \text{ and } \text{ia}(i) \leq p_1, p_2 &lt; \text{ia}(i + 1) \text{ and } i \neq ja(p_1) \text{ and } i \neq ja(p_2)})</td>
</tr>
<tr>
<td>D10</td>
<td>2: (u'[i]) (read)</td>
<td>2: (u'[ja[p]])</td>
<td>read-read dependence</td>
</tr>
<tr>
<td>D11</td>
<td>2: (u'[i]) (read)</td>
<td>4: (u'[i]) (read)</td>
<td>read-read dependence</td>
</tr>
<tr>
<td>D12</td>
<td>2: (u'[i]) (read)</td>
<td>4: (u'[i]) (write)</td>
<td>({[\text{iter}_1, i, p] \rightarrow [\text{iter}_2, i] \mid 1 \leq \text{iter}_1 &lt; \text{iter}_2 \leq T \text{ and } 0 \leq i &lt; R \text{ and } \text{ia}(i) \leq p &lt; \text{ia}(i + 1) \text{ and } i \neq ja(p)})</td>
</tr>
<tr>
<td>D13</td>
<td>2: (u'[i]) (write)</td>
<td>1: (u'[i])</td>
<td>same as dependence relation D7</td>
</tr>
</tbody>
</table>

Table 2.1: Data dependence relations for Gauss-Seidel CSR - Part I
<table>
<thead>
<tr>
<th>ID</th>
<th>Ref 1</th>
<th>Ref 2</th>
<th>Data dependence relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D14</td>
<td>2: $u'[i]$ (write)</td>
<td>2: $u'[i]$ (read)</td>
<td>same as dependence relation D9</td>
</tr>
<tr>
<td>D15</td>
<td>2: $u'[i]$ (write)</td>
<td>2: $u'[i]$ (write)</td>
<td>same as dependence relation D9</td>
</tr>
</tbody>
</table>
| D16 | 2: $u'[i]$ (write) | 2: $u'[ja[p]]$ (write) | \{[iter, i_1, p_1] → [iter, i_2, p_2] \mid 1 \leq iter \leq T \\
and 0 \leq i_1 < i_2 < R \text{ and } ia(i_1) < p_1 < ia(i_1 + 1) \text{ and } i_1 \neq ja(p_1) \text{ and } ia(i_2) \leq p_2 < ia(i_2 + 1) \text{ and } i_2 \neq ja(p_2) \text{ and } p_1 = ja(i_2) \}
\cup
\{[iter, i_1, p_1] → [iter, i_2, p_2] \mid 1 \leq iter_1 < iter_2 \leq T \text{ and } 0 \leq i_1, i_2 < R \text{ and } ia(i_1) \leq p_1 < ia(i_1 + 1) \text{ and } i_1 \neq ja(p_1) \text{ and } ia(i_2) < p_2 < ia(i_2 + 1) \text{ and } i_2 \neq ja(p_2) \text{ and } p_2 = ja(i_2) \}
\}
| D17 | 2: $u'[i]$ (write) | 4: $u'[i]$ (read) | same as dependence relation D12 |
| D18 | 2: $u'[i]$ (write) | 4: $u'[i]$ (write) | same as dependence relation D12 |
| D19 | 2: $u'[ja[p]]$ | 1: $u'[i]$ | \{[iter, i_1, p] → [iter, i_2] \mid 1 \leq iter \leq T \\
and 0 \leq i_1 < i_2 < R \text{ and } ia(i_1) \leq p < ia(i_1 + 1) \text{ and } i_1 \neq ja(p) \text{ and } ja(p) = i_2 \}
\cup
\{[iter_1, i_1, p] → [iter_2, i_2] \mid 1 \leq iter_1 < iter_2 \leq T \text{ and } 0 \leq i_1, i_2 < R \text{ and } ia(i_1) \leq p < ia(i_1 + 1) \text{ and } i_1 \neq ja(p) \text{ and } ja(p) = i_2 \}
| D20 | 2: $u'[ja[p]]$ | 2: $u'[i]$ (read) | read-read dependence |
| D21 | 2: $u'[ja[p]]$ (write) | 2: $u'[i]$ | \{[iter, i_1, p_1] → [iter, i_2, p_2] \mid 1 \leq iter \leq T \\
and 0 \leq i_1 < i_2 < R \text{ and } ia(i_1) < p_1 < ia(i_1 + 1) \text{ and } i_1 \neq ja(p_1) \text{ and } ia(i_2) \leq p_2 < ia(i_2 + 1) \text{ and } i_2 \neq ja(p_2) \text{ and } ja(p_1) = i_2 \}
\cup
\{[iter, i_1, p_1] → [iter, i_2, p_2] \mid 1 \leq iter_1 < iter_2 \leq T \text{ and } 0 \leq i_1, i_2 < R \text{ and } ia(i_1) \leq p_1 < ia(i_1 + 1) \text{ and } i_1 \neq ja(p_1) \text{ and } ia(i_2) \leq p_2 < ia(i_2 + 1) \text{ and } i_2 \neq ja(p_2) \text{ and } ja(p_1) = i_2 \}
| D22 | 2: $u'[ja[p]]$ | 2: $u'[ja[p]]$ | read-read dependence |
| D23 | 2: $u'[ja[p]]$ (read) | 4: $u'[i]$ | read-read dependence |

Table 2.2: Data dependence relations for Gauss-Seidel CSR - Part II
<table>
<thead>
<tr>
<th>ID</th>
<th>Ref 1</th>
<th>Ref 2</th>
<th>Data dependence relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D24</td>
<td>2: (u'[ja[p]])</td>
<td>4: (u'[i])</td>
<td>same as dependence relation D19</td>
</tr>
<tr>
<td></td>
<td>(\text{read}))</td>
<td>(\text{write}))</td>
<td></td>
</tr>
<tr>
<td>D25</td>
<td>4: (u'[i])</td>
<td>1: (u'[i])</td>
<td>same as dependence relation D1</td>
</tr>
<tr>
<td></td>
<td>(\text{read}))</td>
<td>(\text{write}))</td>
<td></td>
</tr>
<tr>
<td>D26</td>
<td>4: (u'[i])</td>
<td>2: (u'[i])</td>
<td>read-read dependence</td>
</tr>
<tr>
<td></td>
<td>(\text{read}))</td>
<td>(\text{read}))</td>
<td></td>
</tr>
<tr>
<td>D27</td>
<td>4: (u'[i])</td>
<td>2: (u'[i])</td>
<td>({\text{iter}_1, i \rightarrow \text{iter}_2, i, p \mid 1 \leq \text{iter}_1 &lt; \text{iter}_2 \leq T \text{ and } 0 \leq i &lt; R \text{ and } ia(i) \leq p &lt; ia(i+1) \text{ and } i \neq ja(p)})</td>
</tr>
<tr>
<td></td>
<td>(\text{read}))</td>
<td>(\text{write}))</td>
<td></td>
</tr>
<tr>
<td>D28</td>
<td>4: (u'[i])</td>
<td>2: (u'[ja[p]])</td>
<td>read-read dependence</td>
</tr>
<tr>
<td></td>
<td>(\text{read}))</td>
<td>(\text{write}))</td>
<td></td>
</tr>
<tr>
<td>D29</td>
<td>4: (u'[i])</td>
<td>4: (u'[i])</td>
<td>read-read dependence</td>
</tr>
<tr>
<td></td>
<td>(\text{read}))</td>
<td>(\text{read}))</td>
<td></td>
</tr>
<tr>
<td>D30</td>
<td>4: (u'[i])</td>
<td>4: (u'[i])</td>
<td>same as dependence relation D5</td>
</tr>
<tr>
<td></td>
<td>(\text{read}))</td>
<td>(\text{write}))</td>
<td></td>
</tr>
<tr>
<td>D31</td>
<td>4: (u'[i])</td>
<td>1: (u'[i])</td>
<td>same as dependence relation D1</td>
</tr>
<tr>
<td></td>
<td>(\text{write}))</td>
<td>(\text{write}))</td>
<td></td>
</tr>
<tr>
<td>D32</td>
<td>4: (u'[i])</td>
<td>2: (u'[i])</td>
<td>same as dependence relation D27</td>
</tr>
<tr>
<td></td>
<td>(\text{write}))</td>
<td>(\text{read}))</td>
<td></td>
</tr>
<tr>
<td>D33</td>
<td>4: (u'[i])</td>
<td>2: (u'[i])</td>
<td>same as dependence relation D27</td>
</tr>
<tr>
<td></td>
<td>(\text{write}))</td>
<td>(\text{write}))</td>
<td></td>
</tr>
<tr>
<td>D34</td>
<td>4: (u'[i])</td>
<td>2: (u'[ja[p]])</td>
<td>same as dependence relation D4</td>
</tr>
<tr>
<td></td>
<td>(\text{write}))</td>
<td>(\text{write}))</td>
<td></td>
</tr>
<tr>
<td>D35</td>
<td>4: (u'[i])</td>
<td>4: (u'[i])</td>
<td>same as dependence relation D1</td>
</tr>
<tr>
<td></td>
<td>(\text{write}))</td>
<td>(\text{read}))</td>
<td></td>
</tr>
<tr>
<td>D36</td>
<td>4: (u'[i])</td>
<td>4: (u'[i])</td>
<td>same as dependence relation D1</td>
</tr>
<tr>
<td></td>
<td>(\text{write}))</td>
<td>(\text{write}))</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: Data dependence relations for Gauss-Seidel CSR Part III

dependences due to the diag array do not affect the legality of reordering the i loop, and therefore are not included in the list of data dependence relations.

### 2.3.2 Compile-time Transformation Mappings

The sparse-tiled Gauss-Seidel code in Figure 2.6 executes points in the iteration space \({[\text{iter}, i] \mid (1 \leq \text{iter} \leq T) \text{ and } (0 \leq i < R)}\) in a tile-by-tile
fashion. To generate the sparse-tiled Gauss-Seidel executor, the code in Figure 2.5 is transformed to a unified iteration space that includes a tiling loop. Ahmed et al. [5] and Kelly-Pugh [73] give two different methods for building a unified iteration space. In this dissertation, we use the Kelly-Pugh method. In a unified iteration space, each loop that a statement resides within corresponds to a pair of dimensions, where the first dimension of the pair is a value of the index variable, and the second dimension is the numerical order of the enclosed loop or statement. For Figure 2.6, the unified iteration space can be described as follows:

\[
\begin{align*}
\{[\text{tileID}, 1, \text{iter}, 1, i, 1, 1, 1] & \mid (1 \leq \text{iter} \leq T) \text{ and } (0 \leq i < R) \} \\
\cup \{[\text{tileID}, 1, \text{iter}, 1, i, 2, p, s] & \mid (1 \leq \text{iter} \leq T) \text{ and } (0 \leq i < R) \\
& \quad \text{and } ia(i) \leq p < ia(i + 1) - 1 \text{ and } 1 \leq s \leq 2 \} \\
\cup \{[\text{tileID}, 1, \text{iter}, 1, i, 3, 1, 1] & \mid (1 \leq \text{iter} \leq T) \text{ and } (0 \leq i < R) \}.
\end{align*}
\]

Lexicographical order on integer tuples can be defined as follows [72]: writing \((x_1, ..., x_n) \prec (y_1, ..., y_n)\) means there exists an \(m\) such that \((\forall i : 1 \leq i < m \Rightarrow x_i = y_i)\) and \((x_m < y_m)\). The unified iteration space is executed in lexicographical order by the code in Figure 2.6: by tile, by convergence iteration, and then by the order specified with the data reordering function \(\sigma\) since \(i = \sigma(v)\).

The following integer space mappings [73] describe the code transformation from the code in Figure 2.5 to that in Figure 2.6. Transformation mapping \(M_s\) maps Statement \(s\) from its original iteration space to a transformed unified iteration space:

\[
\begin{align*}
M_1 &= \{[\text{iter}, i] \rightarrow [\theta(\text{iter}, \sigma^{-1}(i)), 1, \text{iter}, 1, i, 1, 1, 1]\} \\
M_2 &= \{[\text{iter}, i, p] \rightarrow [\theta(\text{iter}, \sigma^{-1}(i)), 1, \text{iter}, 1, i, 2, p, 1]\} \\
M_3 &= \{[\text{iter}, i, p] \rightarrow [\theta(\text{iter}, \sigma^{-1}(i)), 1, \text{iter}, 1, i, 2, p, 2]\} \\
M_4 &= \{[\text{iter}, i] \rightarrow [\theta(\text{iter}, \sigma^{-1}(i)), 1, \text{iter}, 1, i, 3, 1, 1]\}.
\end{align*}
\]
2.3.3 Constraints on the Tiling Function Due to Data Dependences

In this section we use the data dependence relations derived for Figure 2.5 and the transformation mappings that map the code in Figure 2.5 to the code in Figure 2.6, to derive the necessary constraints on the tiling function $\theta$. A transformation mapping is legal if after applying the mapping to each data dependence relation, the new data dependence relation is satisfied by a lexicographical execution of the transformed unified iteration space. If there is a dependence between iteration $\bar{x}$ of Statement $s$ and iteration $\bar{y}$ of Statement $q$ then the transformation mappings must satisfy the transformed dependence [73]. This can be expressed mathematically as follows:

$$\forall s, q, \bar{x}, \bar{y}: \text{ if } \bar{x} \rightarrow \bar{y} \in d_{sq} \text{ then } M_s(\bar{x}) < M_q(\bar{y}),$$

where $d_{sq}$ is the set of data dependences between Statement $s$ and $q$ and $<$ is the lexicographic ordering operator.

For example, the dependences between Statement 1 of Figure 2.5 and itself are as follows:

$$d_{11} = D1 = \{[\text{iter}_1, i] \rightarrow [\text{iter}_2, i] \mid (1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \text{ and } (0 \leq i < R)\}.$$ 

Applying $M_1$ to both the source and the target of the data dependence relation $d_{11}$ results in the following constraint:

$$\forall \text{iter}_1, \text{iter}_2, i \text{ such that } (1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \text{ and } (0 \leq i < R)$$

it follows that

$$[\theta(\text{iter}_1, \sigma^{-1}(i)), 1, \text{iter}_1, 1, i, 1, 1, 1] < [\theta(\text{iter}_2, \sigma^{-1}(i)), 1, \text{iter}_2, 1, i, 1, 1, 1].$$

By using the definition of lexicographical order, the above constraint simplifies to Constraint (2.2).
∀ \text{iter}_1, \text{iter}_2, i \text{ such that } (1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \\
\text{and } (0 \leq i < R) \text{ it follows that} \\
θ(\text{iter}_1, σ^{-1}(i)) \leq θ(\text{iter}_2, σ^{-1}(i)) \quad (2.2)

The tiling function \( θ \) must satisfy Constraint (2.2).

The dependences from Statement 1 to Statement 2 result in a more complex example.

\[ d_{12} = D2 \cup D4 \]

\[ = \{ [\text{iter}_1, i] \rightarrow [\text{iter}_2, i, p] \mid (1 \leq \text{iter}_1 \leq \text{iter}_2 \leq T) \text{ and } (0 \leq i < R) \]

\[ \text{and } (ia(i) \leq p < ia(i + 1)) \text{ and } (i \neq ja(p)) \} \]

\[ \cup \{ [\text{iter}, i_1] \rightarrow [\text{iter}, i_2, p] \mid (1 \leq \text{iter} \leq T) \text{ and } (0 \leq i_1 < i_2 < R) \]

\[ \text{and } (ia(i_2) \leq p < ia(i_2 + 1)) \text{ and } (i_1 = ja(p)) \} \]

\[ \cup \{ [\text{iter}_1, i_1] \rightarrow [\text{iter}_2, i_2, p] \mid (1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \text{ and } (i_1 \neq i_2) \text{ and } \]

\[ (0 \leq i_1, i_2 < R) \text{ and } (ia(i_2) \leq p < ia(i_2 + 1)) \text{ and } (i_1 = ja(p)) \} \]

Applying \( M_1 \) to the source of the dependence relations and \( M_2 \) to the target in each of the three components results in three constraints. The first constraint is

\[ \forall \text{iter}_1, \text{iter}_2, i, p \text{ such that } (1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \text{ and } (0 \leq i < R) \]

\[ \text{and } (ia(i) \leq p < ia(i + 1)) \text{ and } (i \neq ja(p)) \text{ it follows that} \]

\[ [θ(\text{iter}_1, σ^{-1}(i)), 1, \text{iter}_1, 1, i, 1, 1, 1] < [θ(\text{iter}_2, σ^{-1}(i)), 1, \text{iter}_2, 1, i, 2, p, 1]. \]

By applying the definition of lexicographical order, the first constraint simplifies to the following, which is subsumed by the previously determined Constraint (2.2):

\[ \forall \text{iter}_1, \text{iter}_2, i, p \text{ such that } (1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \text{ and } (0 \leq i < R) \]

\[ \text{and } (ia(i) \leq p < ia(i + 1)) \text{ and } (i \neq ja(p)) \text{ it follows that} \]

\[ θ(\text{iter}_1, σ^{-1}(i)) \leq θ(\text{iter}_2, σ^{-1}(i)). \]
The second constraint is

\[ \forall \text{iter}, i_1, i_2, p \text{ such that } (1 \leq \text{iter} \leq T) \text{ and } (0 \leq i_1 < i_2 < R) \]
and \((ia(i_2) \leq p < ia(i_2 + 1))\) and \((i_1 = ja(p))\) it follows that
\[ [\theta(\text{iter}, \sigma^{-1}(i_1)), \text{iter}, i_1, 1, 1, 1] < [\theta(\text{iter}, \sigma^{-1}(i_2)), \text{iter}, i_2, 2, p, 1]. \]

By applying the definition of lexicographical order, the second constraint simplifies to Constraint (2.3).

\[ \forall \text{iter}, i_1, i_2, p \text{ such that } (1 \leq \text{iter} \leq T) \text{ and } (0 \leq i_1 < i_2 < R) \]
and \((ia(i_2) \leq p < ia(i_2 + 1))\) and \((i_1 = ja(p))\)

\[ \text{it follows that } \theta(\text{iter}, \sigma^{-1}(i_1)) \leq \theta(\text{iter}, \sigma^{-1}(i_2)) \]
(2.3)

Finally, the third constraint is

\[ \forall \text{iter}_1, \text{iter}_2, i_1, i_2, p \text{ such that } \]
\((1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \text{ and } (i_1 \neq i_2) \text{ and } (0 \leq i_1, i_2 < R) \]
and \((ia(i_2) \leq p < ia(i_2 + 1))\) and \((i_1 = ja(p))\) it follows that
\[ [\theta(\text{iter}_1, \sigma^{-1}(i_1)), \text{iter}_1, i_1, 1, 1, 1] < [\theta(\text{iter}_2, \sigma^{-1}(i_2)), \text{iter}_2, i_2, 2, p, 1]. \]

By applying the definition of lexicographical order, the third constraint simplifies to Constraint (2.4).

\[ \forall \text{iter}_1, \text{iter}_2, i_1, i_2, p \text{ such that } \]
\((1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \text{ and } (i_1 \neq i_2) \text{ and } (0 \leq i_1, i_2 < R) \]
and \((ia(i_2) \leq p < ia(i_2 + 1))\) and \((i_1 = ja(p))\) it follows that \(\theta(\text{iter}_1, \sigma^{-1}(i_1)) \leq \theta(\text{iter}_2, \sigma^{-1}(i_2))\)
(2.4)

To satisfy the transformed dependences between Statement 1 and Statement 2, the tiling function \(\theta\) must satisfy Constraints (2.2), (2.3), and (2.4).

The remaining constraints on the tiling function \(\theta\) are generated by applying the transformation mappings to all the data dependence relations in...
Tables 2.1 through 2.3 and simplifying the resulting constraints. To verify our results, we apply the mappings to the data dependence relations in the Omega Calculator [71]. We remove uninterpreted function symbols from the mappings and data dependence relations by substituting them with free variables or symbolic constants. This is necessary due to the limitations involving compositions of relations containing uninterpreted function symbols described in the Omega Calculator user manual [71].

**Theorem 1** For a given ordering function $\sigma$ such that $i = \sigma(v)$ and tiling function $\theta$ with the schedule function, $\text{sched(tileID, iter)} = \{ \sigma(v) \mid \theta(\text{iter}, v) = \text{tileID} \}$, the sparse-tiled Gauss-Seidel code shown in Figure 2.6 is a legal transformation of the code of Figure 2.5 if the following constraints are met:

1. $\forall \text{iter}_1, \text{iter}_2, i$ such that $(1 \leq \text{iter}_1 < \text{iter}_2 \leq T)$ and $(0 \leq i < R)$ it follows that $\theta(\text{iter}_1, \sigma^{-1}(i)) \leq \theta(\text{iter}_2, \sigma^{-1}(i))$

2. $\forall \text{iter}, i_1, i_2$ such that $(1 \leq \text{iter} \leq T)$ and $(0 \leq i_1 < i_2 < R)$, if $(\exists p : \text{ia}(i_1) \leq p < \text{ia}(i_1 + 1) \text{ and } i_2 = ja(p))$, then $\theta(\text{iter}, \sigma^{-1}(i_1)) \leq \theta(\text{iter}, \sigma^{-1}(i_2))$

3. $\forall \text{iter}, i_1, i_2$ such that $(1 \leq \text{iter} \leq T)$ and $(0 \leq i_1 < i_2 < R)$, if $(\exists p : \text{ia}(i_2) \leq p < \text{ia}(i_2 + 1) \text{ and } i_1 = ja(p))$, then $\theta(\text{iter}, \sigma^{-1}(i_1)) \leq \theta(\text{iter}, \sigma^{-1}(i_2))$

4. $\forall \text{iter}_1, \text{iter}_2, i_1, i_2$ such that $(1 \leq \text{iter}_1 < \text{iter}_2 \leq T)$ and $(0 \leq i_1 \neq i_2 < R)$, if $(\exists p : \text{ia}(i_1) \leq p < \text{ia}(i_1 + 1) \text{ and } i_2 = ja(p))$, then $\theta(\text{iter}_1, \sigma^{-1}(i_1)) \leq \theta(\text{iter}_2, \sigma^{-1}(i_2))$

5. $\forall \text{iter}_1, \text{iter}_2, i_1, i_2$ such that $(1 \leq \text{iter}_1 < \text{iter}_2 \leq T)$ and $(0 \leq i_1 \neq i_2 < R)$, if $(\exists p : \text{ia}(i_2) \leq p < \text{ia}(i_2 + 1) \text{ and } i_1 = ja(p))$, then $\theta(\text{iter}_1, \sigma^{-1}(i_1)) \leq \theta(\text{iter}_2, \sigma^{-1}(i_2))$

**Proof:** The constraints are generated by applying the transformation mappings $M_1$ through $M_4$ to the data dependence relations for Gauss-Seidel for CSR in
Figure 2.5. For example, the derivation of Constraint 1 in the theorem is the same as that of Constraint (2.2).

Intuitively, Constraint 1 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 that share an edge in the matrix graph must be executed in the order provided by \( \sigma \). Constraints 4 and 5 arise because for unknowns that 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.

### 2.4 Sparse Tiling Inspector for Gauss-Seidel

The sparse tiling inspector subdivides the Gauss-Seidel iteration space into tiles to perform run-time data and iteration reordering. Recall from Section 2.1 that the Gauss-Seidel computation can be visualized as shown in Figure 2.1. The input to the sparse tiling inspector is the number of convergence iterations in the computation and the matrix graph for any square sparse matrix. The inspector generates a data reordering function \( \sigma \), a tiling function \( \theta \), and a schedule function \( sched \). The sparse tiling inspector for a serial execution of sparse-tiled Gauss-Seidel includes the following steps:

- **Partition** the matrix graph to create a seed partitioning.
- **Grow tiles** from the cells of the seed partitioning to create a tiling function \( \theta \), which maps each iteration point to a tile.
- **Generate** the reordering function \( \sigma \).
- **Remap** the data using the reordering function \( \sigma \).
• **Reschedule** by creating a schedule function \( \text{sched} \), based on the tiling function \( \theta \).

The sparse tiling inspector operates on the original matrix graph, which is represented as a set of vertices and edges \( G(V, E) \). As described in Section 2.1, for each iteration \( v \), there is a node in the matrix graph \( v \in V \), and for each nonzero in the matrix, \( a_{vw} \neq 0 \), there is a directed edge \( <v, w> \in E \). The sparse tiling algorithm does not assume that the sparse matrix is symmetric in structure. When calculating the complexity of the various algorithms that are part of the inspector, we assume that the matrix graph is stored in the CSR sparse matrix format. In particular, accessing all edges \( <v, \ast> \), with a given node \( v \) as the first endpoint can be done efficiently (ie. \( O(|w|) \) such that \( <v, w> \in E \)).

The next subsections describe each step of the run-time process for **full sparse tiling** (previously referred to as serial sparse tiling [119]) in terms of algorithmic complexity and provable characteristics of the resulting \( \sigma \) and \( \theta \) functions. The only difference between full sparse tiling and cache blocking [35] is the tile-growth algorithm. Characteristics of the resulting \( \sigma \) and \( \theta \) functions are determined by analyzing the axiomatic semantics [58] of the inspector algorithms. These characteristics are later used in Section 2.4.6 to prove that full sparse-tiled Gauss-Seidel generates results bit-equivalent to typical Gauss-Seidel when the same data order is used by both.

### 2.4.1 Partition the Matrix Graph

Although optimal graph partitioning is an NP-Hard problem [42], there are many heuristics to calculate reasonable graph partitionings. 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. Heuristics are used to perform partitionings in practice. For the results
reported later in this chapter, we use the Metis package [69] to generate the seed partitioning function \( \text{part}(\cdot) \). 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 nodes in the matrix graph \( v \in V \) have been assigned a seed partition \( \text{part}(v) \).

2.4.2 Sparse Tile the Iteration Space

The matrix graph partitioning, generated in the \textbf{Partition} step, creates a seed partitioning from which tiles can be grown. The seed partitioning determines the tiling at a particular convergence iteration, \( \text{iter}_s \). Specifically at \( \text{iter}_s \), where \( 1 \leq \text{iter}_s \leq T \), the tiling function is set to the partition function, \( \theta(\text{iter}_s, v) = \text{part}(v) \). To determine the tiling at other convergence iterations, the tile growth algorithm adds or deletes nodes from the seed partition as needed to ensure that atomic execution of each tile does not violate any data dependences.

The \texttt{FullSparseNaive_GSCSR} algorithm, shown in Figure 2.8, generates the tiling function \( \theta \), which maps iteration points to tiles. While generating the tiling function, ordering constraints between nodes in the matrix graph are maintained in the relation \texttt{NodeOrd}. The first two statements in the algorithm initialize the \texttt{NodeOrd} relation and all of the tiling function values for the convergence iteration \( \text{iter}_s \). The algorithm then loops through the convergence iterations that come before \( \text{iter}_s \) setting \( \theta \) at each iteration point \(<\text{iter}, v>\) to tiling function’s value for the iteration point directly above it in the iteration space. Finally, it visits the edges that have endpoints in two different partition cells, adjusting the tiling function \( \theta \) to ensure that the data dependences are satisfied. The process is repeated for the convergence iterations between \( \text{iter}_s \) and \( T \) in the upward tile growth. Once neighboring nodes, \(<v, w>\in E\), are put into two different tiles at any iteration \( \text{iter} \), the relative order between these two nodes
Algorithm `FullSparseNaive_GSCSR(G(V,E),part(),T,iter_s)`

1: foreach vertex v ∈ V, θ(iter_s, v) ← part(v)  
2: `NodeOrd` ← \{<v, w> | θ(iter_s, v) < θ(iter_s, w)  
and (<v, w> ∈ E or <w, v> ∈ E)\}

Downward tile growth
3: for iter = (iter_s - 1) downto 1  
4:  foreach vertex v ∈ V, θ(iter, v) ← θ(iter + 1, v)  
5:  do while θ changes  
6:   foreach <v, w> ∈ `NodeOrd`  
7:    θ(iter, w) ← min(θ(iter, w), θ(iter + 1, v))  
8:    θ(iter, v) ← min(θ(iter, v), θ(iter, w))  
9:  end foreach  
10: end do while  
11: `NodeOrd` ← `NodeOrd` ∪  
    \{<v, w> | θ(iter, v) < θ(iter, w)  
and (<v, w> ∈ E or <w, v> ∈ E)\}

Upward tile growth
13: for iter = (iter_s + 1) to T  
14:  foreach vertex v ∈ V, θ(iter, v) ← θ(iter - 1, v)  
15:  do while θ changes  
16:   foreach <v, w> ∈ `NodeOrd`  
17:    θ(iter, v) ← max(θ(iter, v), θ(iter - 1, w))  
18:    θ(iter, w) ← max(θ(iter, w), θ(iter, v))  
19:  end foreach  
20: end do while  
21: `NodeOrd` ← `NodeOrd` ∪  
    \{<v, w> | θ(iter, v) < θ(iter, w)  
and (<v, w> ∈ E or <w, v> ∈ E)\}

Figure 2.8: FullSparseNaive_GSCSR Algorithm
must be maintained. The NodeOrd relation maintains that relative order. For example, if \( \theta(\text{iter}, v) < \theta(\text{iter}, w) \) then \( <v, w> \in \text{NodeOrd} \).

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 which begin at lines 5 and 15. In the worst case, the while loop will execute \( k|V| \) times, with only one \( \theta(\text{iter}, v) \) value decreasing (or increasing in the forward tile growth) each time through the while loop. Each \( \theta(\text{iter}, v) \) can take on values between 1 and \( k \), where \( k \) is the number of tiles. In practice, the algorithm runs much faster than this bound.

Figures 2.9 and 2.10 list post-conditions and post-conditions for the FullSparseNaiveGCSR algorithm. Next we give a proof for each of the post-conditions.

**Post-condition 2.1** \( \forall v \in V, \theta(\text{iter}_s, v) \) is initialized

Satisfied by assignment in line 1.  

**Post-condition 2.2** \( \forall v, w \in V, <v, w> \in \text{NodeOrd} \) if and only if
\( \theta(\text{iter}_s, v) < \theta(\text{iter}_s, w) \) and \( <v, w> \in E \) or \( <w, v> \in E \)

Satisfied by assignment in line 2.

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

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

**Post-condition 7.1** \( \theta(\text{iter}, w) \leq \theta(\text{iter} + 1, w) \)
Algorithm FullSparseNaive\(_{-}\)GCSR\((G(V, E), \text{part}(), T, \text{iter}_s)\):

1: \textbf{foreach} vertex \(v \in V\), \(\theta(\text{iter}_s, v) \leftarrow \text{part}(v)\)
2: \(\text{NodeOrd} \leftarrow \{<v, w> \mid \theta(\text{iter}_s, v) < \theta(\text{iter}_s, w)\)
   \quad \text{and} \ (<v, w> \in E \text{ or } <w, v> \in E)\}\)
   \{ post-cond [2.1] \(\forall v \in V, \theta(\text{iter}_s, v)\) is initialized \}
   \{ post-cond [2.2] \(\forall v, w \in V, <v, w> \in \text{NodeOrd}\) if and only if
   \(\theta(\text{iter}_s, v) < \theta(\text{iter}_s, w)\) and \(<v, w> \in E \text{ or } <w, v> \in E\) \}

Downward tile growth
3: \textbf{for} \(\text{iter} = (\text{iter}_s - 1)\) \textbf{down to} 1
   \{ pre-cond [4.1] \(\forall v \in V, \theta(\text{iter} + 1, v)\) is initialized \}
4: \textbf{foreach} vertex \(v \in V\), \(\theta(\text{iter}, v) \leftarrow \theta(\text{iter} + 1, v)\)
   \{ post-cond [4.1] \(\forall v \in V, \theta(\text{iter}, v) = \theta(\text{iter} + 1, v)\) \}
5: \textbf{do while} \(\theta\) changes
6: \textbf{foreach} \(<v, w> \in \text{NodeOrd}\)
7: \(\theta(\text{iter}, w) \leftarrow \min(\theta(\text{iter}, w), \theta(\text{iter} + 1, w))\)
   \{ post-cond [7.1] \(\theta(\text{iter}, w) \leq \theta(\text{iter} + 1, w)\) \}
   \{ post-cond [7.2] \(\theta(\text{iter}, w) \leq \theta(\text{iter} + 1, w)\) \}
8: \(\theta(\text{iter}, v) \leftarrow \min(\theta(\text{iter}, v), \theta(\text{iter}, w))\)
   \{ post-cond [8.1] \(\theta(\text{iter}, v) \leq \theta(\text{iter} + 1, v)\) \}
   \{ post-cond [8.2] \(\theta(\text{iter}, v) \leq \theta(\text{iter}, w)\) \}
9: \textbf{end foreach}
   \{ post-cond [9.1] \(\forall v \in V, \theta(\text{iter}, v) \leq \theta(\text{iter} + 1, v)\) \}
   \{ post-cond [9.2] if \(\theta\) didn’t change then
   \(\forall <v, w> \in \text{NodeOrd},\)
   \(\theta(\text{iter}, v) \leq \theta(\text{iter}, w) \leq \theta(\text{iter} + 1, v)\) \}
10: \textbf{end do while}
   \{ post-cond [10.1] \(\forall <v, w> \in \text{NodeOrd},\)
   \(\theta(\text{iter}, v) \leq \theta(\text{iter}, w) \leq \theta(\text{iter} + 1, v)\) \}
   \{ post-cond [10.2] \(\forall v, w \in V, \text{ if } (<v, w> \in E \text{ or } <w, v> \in E)\)
   \(\text{then } \theta(\text{iter}, v) \leq \theta(\text{iter} + 1, w)\) \}

Figure 2.9: FullSparseNaive\(_{-}\)GCSR with post-conditions, Part I
Algorithm FullSparseNaive_GSCSR cont...

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

12: end for

\{ post-cond [12.1] \( \forall \text{iter} : 1 \leq \text{iter} \leq (\text{iter}_s - 1) \) and \( \forall v \in V, \theta(\text{iter}, v) \leq \theta(\text{iter} + 1, v) \) \}

\{ post-cond [12.2] \( \forall v, w \in V, <v, w> \in \text{NodeOrd} \) if and only if \( \exists \text{iter} : 1 \leq \text{iter} \leq (\text{iter}_s - 1) \) such that \( \theta(\text{iter}, v) < \theta(\text{iter}, w) \) and \( <v, w> \in E \) or \( <w, v> \in E \) \}

\{ post-cond [12.3] \( \forall \text{iter} : 1 \leq \text{iter} \leq (\text{iter}_s - 1) \) and \( \forall v, w \in V, \) if \( <v, w> \in E \) or \( <w, v> \in E \) then \( \theta(\text{iter}, v) \leq \theta(\text{iter} + 1, w) \) \}

\{ post-cond [12.4] \( \text{NodeOrd} \) is acyclic \}

Upward tile growth
13: for \( \text{iter} = (\text{iter}_s + 1) \) to \( T \)

\{ pre-cond [14.1] \( \forall v \in V, \theta(\text{iter} - 1, v) \) is initialized \}

14: foreach vertex \( v \in V, \theta(\text{iter}, v) \leftarrow \theta(\text{iter} - 1, v) \)

\{ post-cond [14.1] \( \forall v \in V, \theta(\text{iter}, v) = \theta(\text{iter} - 1, v) \) \}

15: do while \( \theta \) changes

16: foreach \( <v, w> \in \text{NodeOrd} \)

17: \( \theta(\text{iter}, v) \leftarrow \max(\theta(\text{iter}, v), \theta(\text{iter} - 1, w)) \)

\{ post-cond [17.1] \( \theta(\text{iter} - 1, v) \leq \theta(\text{iter}, v) \) \}

\{ post-cond [17.2] \( \theta(\text{iter} - 1, w) \leq \theta(\text{iter}, v) \) \}

18: \( \theta(\text{iter}, w) \leftarrow \max(\theta(\text{iter}, w), \theta(\text{iter}, v)) \)

\{ post-cond [18.1] \( \theta(\text{iter} - 1, w) \leq \theta(\text{iter}, w) \) \}

\{ post-cond [18.2] \( \theta(\text{iter}, v) \leq \theta(\text{iter}, w) \) \}

19: end foreach

\{ post-cond [19.1] \( \forall v \in V, \theta(\text{iter} - 1, v) \leq \theta(\text{iter}, v) \) \}

\{ post-cond [19.2] if \( \theta \) didn’t change then \( \forall <v, w> \in \text{NodeOrd}, \) \( \theta(\text{iter} - 1, w) \leq \theta(\text{iter}, v) \leq \theta(\text{iter}, w) \) \}

Figure 2.10: FullSparseNaive_GSCSR with post-conditions, Part II
Algorithm `FullSparseNaive_GSCSR` cont...

20: end do while
{ post-cond [20.1] $\forall <v, w> \in \text{NodeOrd}$,
    $\theta(iter - 1, w) \leq \theta(iter, v) \leq \theta(iter, w)$
{ post-cond [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: $\text{NodeOrd} \leftarrow \text{NodeOrd} \cup$
    $\{<v, w> | \theta(iter, v) < \theta(iter, w)$
    and $(<v, w> \in E$ or $<w, v> \in E)$
}

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

Figure 2.11: `FullSparseNaive_GSCSR` with post-conditions, Part III

Post-condition 4.1 ensures that $\theta(iter, w)$ is initialized 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.

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

Post-condition 4.1 ensures that $\theta(iter, v)$ is initialized 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 $\theta$ didn’t change during the foreach loop then
\[
\forall <v, w> \in \text{NodeOrd}, \theta(\text{iter}, v) \leq \theta(\text{iter}, w) \leq \theta(\text{iter} + 1, v)
\]

Follows immediately from post-conditions 7.2 and 8.2. Notice that it is important that $\theta$ not change during the entire foreach loop for this post-condition to be true. For example, assume that $<v_1, v_2> \in \text{NodeOrd}$ and $<v_2, v_3> \in \text{NodeOrd}$ with $\theta(\text{iter} + 1, v_3) = 0, \theta(\text{iter}, v_1) = \theta(\text{iter} + 1, v_1) > 0$, and $\theta(\text{iter}, v_2) > 0$. If edge $<v_1, v_2>$ is visited first in the foreach loop, then it will still be the case that $\theta(\text{iter}, v_1) > 0$ and $\theta(\text{iter}, v_2) > 0$. However, later in the foreach loop when $<v_2, v_3>$ is visited, $\theta(\text{iter}, v_2)$ will be set equal to 0 (at line 8). It will then be the case that $\theta(\text{iter}, v_1) > \theta(\text{iter}, v_2)$, even though $<v_1, v_2> \in \text{NodeOrd}$. This will be remedied the next time through the foreach loop. 

Post-condition 10.1 \[
\forall <v, w> \in \text{NodeOrd}, \theta(\text{iter}, v) \leq \theta(\text{iter}, w) \leq \theta(\text{iter} + 1, v)
\]

The do while loop in lines 5 through 10 ends when $\theta$ 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 \[
\forall v, w \in V, \text{ if } (<v, w> \in E \text{ or } <w, v> \in E) \text{ then } \theta(\text{iter}, v) \leq \theta(\text{iter} + 1, w)
\]

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

Case 1: If $\theta(\text{iter} + 1, v) \leq \theta(\text{iter} + 1, w)$ then due to post-condition 9.1 the following is true and post-condition 10.2 is satisfied: $\theta(\text{iter}, v) \leq \theta(\text{iter} + 1, v)$.
1, v) ≤ θ(iter + 1, w).

Case 2: If θ(iter + 1, v) > θ(iter + 1, w) then due to post-condition 2.2 when iter = (iter_s – 1) and line 11 when 1 ≤ iter < (iter_s – 1), it is the case that <w, v> ∈ NodeOrd. Since <w, v> ∈ NodeOrd, we can swap the roles of v and w in post-condition 10.1, making the following true to satisfy post-condition 10.2: θ(iter, w) ≤ θ(iter, v) ≤ θ(iter + 1, w).

Post-condition 12.1 ∀ iter : 1 ≤ iter ≤ (iter_s – 1) and ∀v ∈ V, θ(iter, v) ≤ θ(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 ∀ v, w ∈ V, <v, w> ∈ NodeOrd if and only if
∃ iter : 1 ≤ iter ≤ iter_s such that θ(iter, v) < θ(iter, w) and (<v, w> ∈ E or <w, v> ∈ 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 ∀ iter : 1 ≤ iter ≤ (iter_s – 1) and ∀v, w ∈ V, if (<v, w> ∈ E or <w, v> ∈ E) then θ(iter, v) ≤ θ(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 guarantees 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 \( <w, x_0> \ldots <x_n, v> \) in \( \text{NodeOrd} \) 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 \( \theta \)'s values for the nodes in the path for the current value of \( \text{iter} \) at line 11:

\[
\theta(\text{iter}, w) \leq \theta(\text{iter}, x_0) \leq \cdots \leq \theta(\text{iter}, x_n) \leq \theta(\text{iter}, v)
\]  

(2.5)

Due to line 11, if the relation \( <v, w> \) is being added to \( \text{NodeOrd} \) then the following is true:

\[
\theta(\text{iter}, v) < \theta(\text{iter}, w)
\]  

(2.6)

Combining (2.5) and (2.6) results in the contradiction that \( \theta(\text{iter}, v) < \theta(\text{iter}, v) \).

Therefore, it is not possible to add a cycle-causing relation to \( \text{NodeOrd} \).

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

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

**Post-condition 17.1** \( \theta(\text{iter} - 1, v) \leq \theta(\text{iter}, v) \)

Post-condition 14.1 ensures that \( \theta(\text{iter}, v) \) is initialized to \( \theta(\text{iter} - 1, v) \). In line 17, \( \theta(\text{iter}, v) \) can only increase in value, thus postcondition 17.1 holds.

**Post-condition 17.2** \( \theta(\text{iter} - 1, w) \leq \theta(\text{iter}, v) \)

Satisfied by the assignment in line 17.

**Post-condition 18.1** \( \theta(\text{iter} - 1, w) \leq \theta(\text{iter}, w) \)
Post-condition 14.1 ensures that \( \theta(\text{iter}, w) \) is initialized to \( \theta(\text{iter} - 1, w) \). In line 18, \( \theta(\text{iter}, w) \) can only increase in value, thus postcondition 18.1 holds. 

**Post-condition 18.2** \( \theta(\text{iter}, v) \leq \theta(\text{iter}, w) \)

Satisfied by the assignment in line 18. 

**Post-condition 19.1** \( \forall v \in V, \theta(\text{iter} - 1, v) \leq \theta(\text{iter}, v) \)

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

**Post-condition 19.2** If \( \theta \) didn’t change during the entire foreach loop then \( \forall < v, w> \in \text{NodeOrd}, \theta(\text{iter} - 1, w) \leq \theta(\text{iter}, v) \leq \theta(\text{iter}, w) \)

Follows immediately from post-conditions 17.2 and 18.2. Notice that it is important that \( \theta \) not change during the entire foreach loop for this post-condition to be true. For example, assume that \( <w_1, w_2> \in \text{NodeOrd} \) and \( <w_2, w_3> \in \text{NodeOrd} \) with \( \theta(\text{iter} - 1, w_1) = \theta(\text{iter}, w_1) = 4, \theta(\text{iter} - 1, w_2) = \theta(\text{iter}, w_2) < 4, \) and \( \theta(\text{iter}, w_3) < 4. \) 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 \( \theta(\text{iter}, w_2) < 4 \) and \( \theta(\text{iter}, w_3) < 4. \) However, later in the foreach loop when \( <w_1, w_2> \) is visited, \( \theta(\text{iter}, w_2) \) will be set equal to 4 due to line 18. It will then be the case that \( \theta(\text{iter}, w_2) > \theta(\text{iter}, w_3) \) even though \( <w_3, w_3> \in \text{NodeOrd}. \) This will be remedied the next time through the foreach loop. 

**Post-condition 20.1** \( \forall < v, w> \in \text{NodeOrd}, \theta(\text{iter} - 1, w) \leq \theta(\text{iter}, v) \leq \theta(\text{iter}, w) \)

The do while loop in lines 15 through 20 ends when \( \theta \) 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** \( \forall v, w \in V, \text{ if } ( < v, w > \in E \text{ or } < w, v > \in E) \text{ then } \theta(\text{iter} - 1, v) \leq \theta(\text{iter}, w) \)}
For all \( v, w \in V \) with \( \text{iter} = (\text{iter}_s + 1) \), the tiling function values \( \theta(\text{iter} - 1, v) \) and \( \theta(\text{iter} - 1, w) \) are set in line 1. For all \( v, w \in V \) with \( \text{iter} \) such that \( (\text{iter}_s + 1) < i \leq T \), the tiling function values \( \theta(\text{iter} - 1, v) \) and \( \theta(\text{iter} - 1, w) \) are set in the previous iteration of the for loop starting at line 15. The relationship between \( \theta(\text{iter} - 1, v) \) and \( \theta(\text{iter} - 1, w) \) falls under two cases, either \( \theta(\text{iter} - 1, v) \leq \theta(\text{iter} - 1, w) \) or \( \theta(\text{iter} - 1, v) > \theta(\text{iter} - 1, w) \).

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

**Case 2:** If \( \theta(\text{iter} - 1, v) > \theta(\text{iter} - 1, w) \) then due to post-conditions 12.2 when \( \text{iter} = (\text{iter}_s + 1) \) and line 11 if \( (\text{iter}_s + 1) < \text{iter} \leq T \), it is the case that \(<w, v> \in \text{NodeOrd}\). Since \(<w, v> \in \text{NodeOrd}\), we can swap the roles of \( v \) and \( w \) in post-condition 20.1 and make the following true to satisfy post-condition 20.2: \( \theta(\text{iter} - 1, v) \leq \theta(\text{iter}, w) \leq \theta(\text{iter}, v) \).

**Post-condition 22.1** \( \forall q : (\text{iter}_s + 1) \leq q \leq T \) 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** \( \forall q : (\text{iter}_s + 1) \leq q \leq T \) 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 \( \theta \) and the set of ordered pairs `NodeOrd` at the end of the algorithm.

**Lemma 1** Upon completion of the `FullSparseNaive_GSCSR` algorithm, \( \forall \text{iter} : 1 \leq \text{iter} \leq (T - 1) \) and \( \forall v \in V, \theta(\text{iter}, v) \leq \theta(\text{iter} + 1, v) \).
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 \( \theta(\text{iter}, v) \) with \( 1 \leq \text{iter} \leq \text{iter}_s \) and \( v \in V \).

Notice that upon substitution of \( q = \text{iter} + 1 \) in post-condition 22.1 we get the following statement.

\[
\forall \text{iter} : (\text{iter}_s + 1) \leq (\text{iter} + 1) \leq T \text{ and } \forall v \in V,
\theta((\text{iter} + 1) - 1, v) \leq \theta(\text{iter} + 1, v) \quad (2.7)
\]

Rewriting (2.7) we get the following.

\[
\forall \text{iter} : \text{iter}_s \leq \text{iter} \leq (T - 1) \text{ and } \forall v \in V, \theta(\text{iter}, v) \leq \theta(\text{iter} + 1, v) \quad (2.8)
\]

By combining the domains of \( \text{iter} \) in post-condition 12.1 and (2.8) we get post-condition 1. 

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

This condition states that for every ordered pair \(<v, w>\) in the relation \( \text{NodeOrd} \), the tiling function for these nodes at one or more of the convergence iterations satisfies a less than relationship, and if there exists a convergence iteration where the tiling function for neighboring nodes satisfies the less than inequality \( \theta(\text{iter}, v) < \theta(\text{iter}, w) \), then \(<v, w> \in \text{NodeOrd} \). It is 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, \( \forall \text{iter} : 1 \leq \text{iter} \leq (T - 1) \text{ and } \forall v, w \in V, \text{ if } (<v, w> \in E \text{ or } <v, w> \in E) \text{ then } \theta(\text{iter}, v) \leq \theta(\text{iter} + 1, w). \)
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 \( \theta(\text{iter}, v) \) with \( 1 \leq \text{iter} \leq \text{iter}_s \) and \( v \in V \).

Notice that upon substitution of \( q = \text{iter} + 1 \) in post-condition 22.2 we get the following statement.

\[
\forall \text{iter} : (\text{iter}_s + 1) \leq (\text{iter} + 1) \leq T \text{ and } \forall <v, w> \in E, \\
\theta((\text{iter} + 1) - 1, v) \leq \theta(\text{iter} + 1, w)
\]  

(2.9)

Rewriting (2.9) we get the following.

\[
\forall \text{iter} : \text{iter}_s \leq \text{iter} \leq (T - 1) \text{ and } \forall <v, w> \in E, \\
\theta(\text{iter}, v) \leq \theta(\text{iter} + 1, w)
\]  

(2.10)

By combining the domains of \( \text{iter} \) in post-condition 12.3 and (2.10) Lemma 3 is satisfied.

\textbf{Lemma 4} Upon completion of the \textsc{FullSparseNaive_GCSR} algorithm, \textsc{NodeOrd} is acyclic.

Post-condition 12.4 guarantees that \textsc{NodeOrd} is acyclic before the beginning of the for loop which starts at line 13. During the upward tile growth, relations are added to \textsc{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 \textsc{NodeOrd}.

We assume the contrary and then derive a contradiction. Assume there is a path \( <w, x_0> \ldots <x_n, v> \) in the \textsc{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 line 21, the following statement is true about the tiling function \( \theta' \)’s
values for the nodes in the path for the current value of \textit{iter} at line 21:

\[
\theta(\text{iter}, w) \leq \theta(\text{iter}, x_0) \leq \cdots \leq \theta(\text{iter}, x_n) \leq \theta(\text{iter}, v) \quad (2.11)
\]

Due to line 21, if the relation \(< v, w \) is being added to \textit{NodeOrd} then the following is true:

\[
\theta(\text{iter}, v) < \theta(\text{iter}, w) \quad (2.12)
\]

Combining (2.11) and (2.12) result in the contradiction that \( \theta(\text{iter}, v) < \theta(\text{iter}, v) \). Therefore, it is not possible to add a relation \(< v, w \) to \textit{NodeOrd} which will cause a cycle. \( \blacksquare \)

\subsection{Generate the Reordering Function}

We have two goals when ordering the unknowns: satisfy the constraints specified in the \textit{NodeOrd} relation and increase intra-iteration locality. First and foremost, the reordering function must satisfy the \textit{NodeOrd} relation for correctness. Second, to the extent possible, 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.

The \textit{tile vector} \(< \theta(1, v), \ldots, \theta(T, v) \) is a vector of length \( T \) that indicates which tile contains a given node \( v \) at each convergence iteration. 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. 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 probabilistic complexity of Quicksort in this instance is \( O(T|V|\lg|V|) \). A radix sort with complexity \( O(T(|k| + |V|)) \) could also be used. Creating the
reordering function $\sigma$ by sorting the nodes based on their tile vectors results in
the following property.

**Lemma 5** If $\sigma$ is constructed by lexicographically ordering the tile vectors, then
for each $<v, w> \in \text{NodeOrd}$, we have $\sigma(v) < \sigma(w)$

**Proof:**

If $<v, w> \in \text{NodeOrd}$ then by Lemma 2 there exists an $\text{iter}$ such that
$\theta(\text{iter}, v) < \theta(\text{iter}, w)$ and there does not exist an $\text{iter}$ such that $\theta(\text{iter}, w) < \theta(\text{iter}, v)$. Therefore, the tile vector $<\theta(1, v), \ldots, \theta(T, v)>$ lexicographically
precedes the tile vector $<\theta(1, w), \ldots, \theta(T, w)>$. Since $\sigma$ is created with a lexi-
cographical sort of the tile vectors, it follows that $\sigma(v) < \sigma(w)$. ⊡

### 2.4.4 Remap the Data

The reordering function $\sigma$, generated as described in Section 2.4.3, is
used to remap the data from the original unknown array $\tilde{u}$ to a new vector $\tilde{u}'$
such that $\tilde{u}'_{\sigma(v)} = \tilde{u}_v$. Also 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 $O(|V| + |E|)$.

### 2.4.5 Create Schedule

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

### 2.4.6 Proof of Correctness

By using Lemmas 1 through 5, which describe the characteristics of the
tiling function $\theta$ and the reordering function $\sigma$ generated by the full sparse tiling
algorithms, we are able to prove that sparse-tiled Gauss-Seidel in Figure 2.6 generates bit-equivalent results to those generated by Gauss-Seidel for CSR in Figure 2.5 when it uses $\sigma$ for its 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 $\tilde{w}'$, right-hand side $\tilde{f}'$ and sparse matrix $A'$, are such that $i = \sigma(v), A'_{\sigma(v)\sigma(w)} = A_{vw}, u'_{\sigma(v)} = u_v$, and $f'_{\sigma(v)} = f_v$. The nonzero structure of $A'$ is represented with the uninterpreted function symbols $ia$ and $ja$, such that for each $A'_{\sigma(v)\sigma(w)}$ there exists $p$ such that $ia(\sigma(v)) \leq p < ia(\sigma(v) + 1)$ and $\sigma(w) = ja(p)$.

The $\theta$ and $\sigma$ functions, generated by the full sparse tiling inspector using **FULLSPARSENAIVE\_GCSR** as the tile growth algorithm, satisfy the constraints in Theorem 1.

**Proof:**

Since $i = \sigma(v)$ 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. $\forall \ iter_1, \ iter_2, v$ such that $(1 \leq \ iter_1 < \ iter_2 \leq T)$, if $(0 \leq \sigma(v) < R)$, then $\theta(\ iter_1, v) \leq \theta(\ iter_2, v)$

2. $\forall \ iter, v, w$ such that $(1 \leq \ iter \leq T)$, if $(0 \leq \sigma(v) < \sigma(w) < R)$ and $<v, w> \in E$, then $\theta(\ iter, v) \leq \theta(\ iter, w)$

3. $\forall \ iter, v, w$ such that $(1 \leq \ iter \leq T)$, if $(0 \leq \sigma(v) < \sigma(w) < R)$ and $<w, v> \in E$, then $\theta(\ iter_1, v) \leq \theta(\ iter, w)$

4. $\forall \ iter_1, \ iter_2, v, w$ such that $(1 \leq \ iter_1 < \ iter_2 \leq T)$, if $(0 \leq \sigma(v) \neq \sigma(w) < R)$ and $<v, w> \in E$, then $\theta(\ iter_1, v) \leq \theta(\ iter_2, w)$
5. \( \forall \text{iter}_1, \text{iter}_2, v, w \) such that \( 1 \leq \text{iter}_1 < \text{iter}_2 \leq T \), if \( 0 \leq \sigma(v) \neq \sigma(w) < R \) and \( <w, v> \in E \), then \( \theta(\text{iter}_1, v) \leq \theta(\text{iter}_2, w) \)

Constraint 1 is satisfied by transitive closure on Lemma 1.

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

\[
\forall \text{iter}, v, w \text{ such that } (1 \leq \text{iter} \leq T) \text{ and } <v, w> \in E \\
\text{and } (0 \leq \sigma(v) < \sigma(w) < R) \text{ and } \theta(\text{iter}, v) > \theta(\text{iter}, w)
\]

(2.13)

Using Lemma 2 and the assumption from Equation (2.13) that \( \theta(\text{iter}, v) > \theta(\text{iter}, w) \), it must follow that \( <w, v> \in \text{NodeOrd} \). Therefore according to Lemma 5, \( \sigma(w) < \sigma(v) \) which contradicts Equation (2.13).

Constraint 3 can be shown with the same proof that was used for constraint 2. Constraint 4 is satisfied by transitive closure on Lemma 3. Constraint 5 is satisfied by transitive closure on Lemma 3.

2.4.7 Full Sparse Tiling Efficiency Issues

The **Partition, Grow Tiles, Generate reordering, Remap,** and **Reschedule** steps in the sparse tiling inspector account for the run-time overhead of sparse tiling. Since all of these steps occur at runtime, their efficiency is important. The tile growth algorithm **FULLSPARSENAIVE** 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. Figures 2.12 and 2.13 show the tile growth algorithm **FULLSPARSEWORKSET** 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** algorithm the while loop at line 5 is necessary because a specific \( \theta(\text{iter}, v) \) could change multiple
Algorithm FullSparseWorkSet\_GCSR(G(V, E), part(), T, \textit{iter})

1: \forall v \in V \text{ and } 1 \leq \textit{iter} \leq T, \theta(\textit{iter}, v) \leftarrow \text{part}(v)
2: G_{sym}(V_{sym}, E_{sym}) = \text{MakeSymmetricGraph}(G(V, E))
3: NodeOrd \leftarrow \{< v, w > | \theta(\textit{iter}_s, v) < \theta(\textit{iter}_s, w) \text{ and } < v, w > \in E_{sym}\}

Downwards tile growth
4: \forall 0 \leq t < \text{numtile}, \text{TileWorkSet}_t \leftarrow \emptyset
5: \text{for } \textit{iter} = \text{iter}_s - 1 \text{ downto } 1
6: \text{ ThetaChangedWorkSet } \leftarrow \emptyset
7: \text{ foreach } < v, w > \in \text{NodeOrd}
8: \text{ TileWorkSet}_{\theta(\textit{iter}, w)} \leftarrow \text{TileWorkSet}_{\theta(\textit{iter}, w)} \cup \{w\}
9: \text{ if } \theta(\textit{iter}, w) > \theta(\textit{iter} + 1, v) \text{ then UPDATE\_THETA(1, } \textit{iter}, w, \theta(\textit{iter} + 1, v))
10: \text{ end foreach}
11: \text{ for } t = 0 \text{ to } (\text{numtile} - 1)
12: \text{ toCheck } \leftarrow \text{TileWorkSet}_t
13: \text{ while (toCheck } \neq \emptyset)
14: \text{ tempSet } \leftarrow \text{toCheck}; \text{toCheck } \leftarrow \emptyset
15: \text{ foreach } w \in \text{tempSet}; \text{ foreach } < v, w > \in \text{NodeOrd}
16: \text{ if } \theta(\textit{iter}, v) > \theta(\textit{iter}, w) \text{ then}
17: \text{ UPDATE\_THETA(1, } \textit{iter}, v, \theta(\textit{iter}, w))
18: \text{ toCheck } \leftarrow \text{toCheck} \cup \{v\}
19: \text{ end foreach}; \text{ end foreach}
20: \text{ end while}
21: \text{ end for}
22: \text{ NodeOrd } \leftarrow \text{NodeOrd} \cup \{< v, w > | \theta(\textit{iter}, v) < \theta(\textit{iter}, w) \text{ and } v \in \text{ThetaChangedWorkSet} \text{ and } < v, w > \in E_{sym}\}
23: \text{ end for}

Figure 2.12: FullSparseWorkSet\_GCSR Algorithm - Part I

times. Such a change occurs if a relation \(< v, w > \) is in \text{NodeOrd} and \theta(\textit{iter}, w) changes due to a relation \(< w, z > \in \text{NodeOrd} \text{ that is visited later than } < v, w > \text{ in the foreach loop. The FullSparseWorkSet\_GCSR algorithm avoids the need for the while loop by incorporating two changes. First FullSparseWorkSet\_GCSR has two loops, at lines 7 and 11, over the relations in NodeOrd.}
Algorithm **FULLSPARSEWORKSET**\_GSCSR continued ...

Upwards tile growth
24: \( \forall 0 \leq t < \text{numtile}, \text{TileWorkSet}_t \leftarrow \emptyset \)
25: for \( \text{iter} = \text{iter}_s + 1 \) to \( T \)
26: \( \text{ThetaChangedWorkSet} \leftarrow \emptyset \)
27: foreach \( <v, w> \in \text{NodeOrd} \)
28: \( \text{TileWorkSet}_{\theta(\text{iter}, v)} \leftarrow \text{TileWorkSet}_{\text{iter}} \cup \{v\} \)
29: if \( \theta(\text{iter}, v) < \theta(\text{iter} - 1, w) \) then
   \( \text{UPDATETHETA}(\text{iter}, T, v, \theta(\text{iter} - 1, w)) \)
30: end foreach
31: for \( t = (\text{numtile} - 1) \) downto 0
32: \( \text{toCheck} \leftarrow \text{TileWorkSet}_t \)
33: while (\( \text{toCheck} \neq \emptyset \))
34: \( \text{tempSet} \leftarrow \text{toCheck}; \text{toCheck} \leftarrow \emptyset \)
35: foreach \( w \in \text{tempSet} \); foreach \( <v, w> \in \text{NodeOrd} \)
36: if \( \theta(\text{iter}, w) < \theta(\text{iter}, v) \) then
   \( \text{UPDATETHETA}(\text{iter}, T, w, \theta(\text{iter}, v)) \)
37: \( \text{toCheck} \leftarrow \text{toCheck} \cup \{w\} \)
38: end foreach; end foreach
39: end while
40: end for
41: \( \text{NodeOrd} \leftarrow \text{NodeOrd} \cup \{<v, w>|\theta(\text{iter}, v) < \theta(\text{iter}, w)\) and \( w \in \text{ThetaChangedWorkSet} \) and \( <v, w> \in E_{sym} \} \)
42: end for

Figure 2.13: **FULLSPARSEWORKSET**\_GSCSR Algorithm - Part II

The first loop makes sure that if node \( v \) comes before node \( w \) in the \( \text{NodeOrd} \) relation, that the iteration point \( <\text{iter}, w> \) must be in the same or an earlier tile than the iteration point \( <\text{iter} + 1, v> \). Since the tiling function values for all nodes \( v \in V \) won’t change at the \( (\text{iter} + 1) \) iteration, it is only necessary to visit each \( <v, w> \in \text{NodeOrd} \) once to get \( \theta(\text{iter}, w) \leq \theta(\text{iter} + 1, v) \). The second loop through the relations in \( \text{NodeOrd} \) makes sure that if \( <v, w> \in \text{NodeOrd} \) then iteration point \( <\text{iter}, v> \) is put into the same or earlier tile as iteration point
Algorithm MAKESYMMETRICGRAPH($G(V, E)$)

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

Figure 2.14: MAKESYMMETRICGRAPH Algorithm, which is used in FULLSPARSEWORKSET_GSCSR

Algorithm UPDATETHETA(start, end, x, newval)

1: $oldval \leftarrow \theta(\text{start}, x)$
2: $\text{TileWorkSet}_{oldval} \leftarrow \text{TileWorkSet}_{oldval} \setminus \{x\}$
3: $\text{TileWorkSet}_{newval} \leftarrow \text{TileWorkSet}_{newval} \cup \{x\}$
4: $\text{ThetaChangedWorkSet} \leftarrow \text{ThetaChangedWorkSet} \cup \{x\}$
5: $\forall \text{start} \leq q \leq \text{end}, \theta(q, x) \leftarrow \text{newval}$

Figure 2.15: UPDATETHETA Algorithm, which is used in FULLSPARSEWORKSET_GSCSR

$<\text{iter}, w>$. Since we visit the NodeOrd relations $<v, w>$ in order of the current tiling function value for $w$, $\theta(\text{iter}, w)$, at any node $v$ there will not be a path in NodeOrd, $<v, w>, <w, x_1>, ..., <x_{n-1}, x_n>$ such that $\theta(\text{iter}, x_n) < \theta(\text{iter}, w)$. Therefore, it is only necessary to visit each $<v, w> \in \text{NodeOrd}$ 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 $\theta(\text{iter} + 1, v) \geq \theta(\text{iter} + 1, w)$. Thus, it is only necessary to check an edge $<v, w>$ if either $\theta(\text{iter}, v)$ has decreased dur-
ing downward tile growth or $\theta(\text{iter}, w)$ has increased during forward tile growth. In FullSparseWorkSet\_GSCSR, ThetaChangedWorkSet keeps track of all nodes $v$ whose $\theta(\text{iter}, v)$ 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 does not 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 are specified in Lemmas 1-4.

Lemma 6 Upon completion of the FullSparseWorkSet\_GSCSR algorithm, $\forall \text{iter}, v$ such that $1 \leq \text{iter} \leq (T - 1)$ and $v \in V$ it follows that $\theta(\text{iter}, v) \leq \theta(\text{iter} + 1, v)$.

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

Lemma 7 Upon completion of the FullSparseWorkSet\_GSCSR algorithm, $\forall v, w \in V, <v, w> \in \text{NodeOrd}$ if and only if $\exists \text{iter}$ such that $1 \leq \text{iter} \leq T$ and $\theta(\text{iter}, v) < \theta(\text{iter}, w)$ and ($<v, w> \in E$ or $<w, v> \in E$).

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

\[
\forall v, w \in V, \text{ if } \exists \text{iter such that } 1 \leq \text{iter} \leq T \text{ and } \theta(\text{iter}, v) < \theta(\text{iter}, w) \\
\text{and } (<v, w> \in E \text{ or } <w, v> \in E) \text{ then } <v, w> \in NodeOrd.
\]

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

\[
\forall <v, w> \in E_{sym} \text{ such that } 1 \leq \text{iter} \leq \text{iter}_{s}, \\
\text{if } \theta(\text{iter}, v) < \theta(\text{iter}, w) \text{ then } <v, w> \in NodeOrd
\] (2.14)

Proposition (2.14) is true for \(\text{iter} = \text{iter}_{s}\) due to line 3 in the algorithm. Assume the following proposition is true.

\[
\forall <x, y> \in E_{sym} \text{ such that } 1 \leq \text{iter} < \text{iter}_{s}, \\
\text{if } \theta(\text{iter} + 1, x) < \theta(\text{iter} + 1, y) \text{ then } <x, y> \in NodeOrd
\] (2.15)

The following proposition is equivalent to proposition (2.15).

\[
\forall <x, y> \in E_{sym} \text{ such that } 1 \leq \text{iter} < \text{iter}_{s}, \\
\text{if } <x, y> \notin \text{NodeOrd} \text{ then } \theta(\text{iter} + 1, x) \geq \theta(\text{iter} + 1, y)
\] (2.16)

Due to line 1 in the algorithm and the calls to \(\text{UPDATETHETA}\), at the start of the loop in lines 5-23 if \(\theta(\text{iter} + 1, x) \geq \theta(\text{iter} + 1, y)\) then \(\theta(\text{iter}, x) \geq \theta(\text{iter}, y)\). The only way for the inequality to change to \(\theta(\text{iter}, x) < \theta(\text{iter}, y)\) would be if \(\theta(\text{iter}, x)\) is reduced at lines 9 or 17. If that occurs then \(x\) is put in the set \(\text{ThetaChangedWorkSet}\). Due to line 22, if \(x \in \text{ThetaChangedWorkSet}\) and \(\theta(\text{iter}, x) < \theta(\text{iter}, y)\) and \(<x, y> \in E_{sym}\), then \(<x, y>\) is put into the relation \(\text{NodeOrd}\). Therefore (2.14) is true for \(\text{iter}\) such that \(1 \leq \text{iter} < \text{iter}_{s}\).

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

\[
\forall <x, y> \in E_{sym} \text{ such that } \text{iter}_{s} < \text{iter} \leq T,
\]
if \( \theta(\text{iter} - 1, x) < \theta(\text{iter} - 1, y) \) then \( <x, y> \in \text{NodeOrd} \).

- \textbf{Lemma 8} Upon completion of the \textsc{FullSparseWorkSet\_GCSR} algorithm, \( \forall \text{iter}, v, w \) such that \( 1 \leq \text{iter} \leq (T - 1) \) and \( v, w \in V \), if \( (<v, w> \in E \text{ or } <w, v> \in E) \) then \( \theta(\text{iter}, v) \leq \theta(\text{iter} + 1, w) \).

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 \textsc{FullSparseWorkSet\_GCSR} algorithm.

\[
\forall v, w \in V, \text{ if } <v, w> \in \text{Esym} \text{ then } \theta(\text{iter}, v) \leq \theta(\text{iter} + 1, w) \tag{2.17}
\]

\[
\forall v, w \in V, \text{ if } <v, w> \in \text{Esym} \text{ then } \theta(\text{iter} - 1, v) \leq \theta(\text{iter}, w) \tag{2.18}
\]

Proof for (2.17):
This proof is similar in form to that of Post-condition 10.2 from the \textsc{FullSparseNaive\_GCSR} algorithm. If \( \theta(\text{iter} + 1, v) \leq \theta(\text{iter} + 1, w) \), then due to lemma 6, \( \theta(\text{iter}) < \theta(\text{iter} + 1, w) \). For the case where \( \theta(\text{iter} + 1, v) > \theta(\text{iter} + 1, w) \), then due to lemma 7, it is true that \( <w, v> \in \text{NodeOrd} \). Switching \( v \) and \( w \) at line 7, it will be the case that in line 9 there will be an update such that \( \theta(\text{iter}, v) < \theta(\text{iter} + 1, w) \).

Proof for (2.18):
This proof is similar in form to that of Post-condition 20.2 from the \textsc{FullSparseNaive\_GCSR} algorithm. If \( \theta(\text{iter} - 1, v) \leq \theta(\text{iter} - 1, w) \) then due to lemma 6, \( \theta(\text{iter} - 1, v) \leq \theta(\text{iter}, w) \). For the case where \( \theta(\text{iter} - 1, v) > \theta(\text{iter} - 1, w) \) then due to lemma 7, it is true that \( <w, v> \in \text{NodeOrd} \). 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 \( \theta(\text{iter} - 1, v) \leq \theta(\text{iter}, w) \).

\textbf{Lemma 9} Upon completion of the \textsc{FullSparseWorkSet\_GCSR} algorithm, \text{NodeOrd} is acyclic.
<table>
<thead>
<tr>
<th>Matrix</th>
<th>rows</th>
<th>nonzeros</th>
<th>avg nonzeros per row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wing22K</td>
<td>22,266</td>
<td>923,925</td>
<td>41.49</td>
</tr>
<tr>
<td>2D Bar</td>
<td>74,926</td>
<td>1,037,676</td>
<td>13.85</td>
</tr>
<tr>
<td>3D Bar</td>
<td>122,061</td>
<td>4,828,779</td>
<td>39.56</td>
</tr>
<tr>
<td>Sphere150K</td>
<td>154,938</td>
<td>11,508,390</td>
<td>74.28</td>
</tr>
<tr>
<td>Pipe</td>
<td>381,120</td>
<td>15,300,288</td>
<td>40.15</td>
</tr>
<tr>
<td>Wing903K</td>
<td>924,672</td>
<td>38,360,266</td>
<td>41.49</td>
</tr>
</tbody>
</table>

Table 2.4: Summary of input matrices used in experiments for sparse tiling stationary iterative methods

The proof is similar to that of Post-condition 12.4 and Lemma 4. Essentially \textit{NodeOrd} is initialized as acyclic, and at lines 22 and 42 in the \textsc{FullSparseWorkSet\_GCSR} algorithm relation $<v,w>$ is only added to \textit{NodeOrd} when $\theta(\text{iter},v) < \theta(\text{iter},w)$ and $<v,w> \in E_{\text{sym}}$. The loops starting at lines 7, 11, 27, and 31 make sure that if $\exists \textit{iter}_1$ such that $\theta(\text{iter}_1,v) < \theta(\text{iter}_1,w)$, then $\forall \textit{iter}_2 \neq \textit{iter}_1$ it is the case that $\theta(\text{iter}_2,v) \leq \theta(\text{iter}_2,w)$.  

2.5 Matrix Benchmark Suite

In this chapter and Chapters 3 and 4, we use various sparse matrices resulting from Finite Element Analysis of 2D and 3D linear elasticity boundary problems. Table 2.4 summarizes some of the matrix characteristics, and Table 2.5 summarizes some of the characteristics of the Finite Element Analysis. The matrices come from two sources. Mark Adams hosts the Finite Element Market [1] website, which provides matrices generated from test problems that include properties like material coefficient jumps, shell problems, etc. The other matrices were generated using the MC (Manifold Code) software that is part of the FEtk software package [60]. FEtk was developed by Mike Holst and collaborators.

Intuitively, elasticity boundary problems consist of a static object under-
<table>
<thead>
<tr>
<th>Matrix</th>
<th>source</th>
<th>finite elements</th>
<th>elasticity modulus (E)</th>
<th>Poisson ratio (ν)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wing22K</td>
<td>Finite Element Market</td>
<td>four node quadrilateral shell</td>
<td>1</td>
<td>0.3</td>
</tr>
<tr>
<td>2D Bar</td>
<td>FEtk</td>
<td>triangle</td>
<td>21</td>
<td>0.28</td>
</tr>
<tr>
<td>3D Bar</td>
<td>FEtk</td>
<td>tetrahedron</td>
<td>21</td>
<td>0.28</td>
</tr>
<tr>
<td>Sphere150K</td>
<td>Finite Element Market</td>
<td>hexahedron</td>
<td>1 and 10⁻⁴</td>
<td>0.3 and 0.49</td>
</tr>
<tr>
<td>Pipe</td>
<td>FEtk</td>
<td>tetrahedron</td>
<td>21</td>
<td>0.28</td>
</tr>
<tr>
<td>Wing903K</td>
<td>Finite Element Market</td>
<td>four node quadrilateral shell</td>
<td>1</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Table 2.5: Characteristics of Finite Element Analysis problems

going a constant amount of force. For example, in the 2D and 3D Bar problems, the center of the bars are experiencing a load. The goal is to determine the object deformation due to that force. In boundary value problems, constants on the boundaries (Dirichlet boundary conditions), forces applied to the boundaries (Neumann boundary conditions), and the partial differential equations that hold within the domain, are all discretized over finite elements to generate a system of linear equations.

The finite elements used for these problems are the triangle, the tetrahedron (3D generalization of the triangle), the quadrilateral shell, and the hexahedron (3D generalization of a quadrilateral). The Young’s modulus and the Poisson ratio are numerical constants describing the relationship between stress to strain within various materials.

In our experiments involving sparse-tiled stationary iterative methods, the most important characteristics of these matrices are their size and the ratio of nonzeros to rows. This is because the sparse tiling seed partitioning and tile
Table 2.6: Descriptions of computer architectures used in the serial experiments

growth algorithms operate on the matrix graph. The number of rows in the
matrix corresponds to the number of nodes in the matrix graph. The ratio of
nonzeros to rows indicates the average number of edges coming out of each node.

Even though our full sparse tile growth algorithm handles general square
sparse matrices, all of these matrices are symmetric positive definite. A matrix
is symmetric if for all rows and columns, $A_{rc} = A_{cr}$. A matrix is positive definite
if for all $\bar{x} \in \mathbb{R}^n$, $\bar{x}^T A \bar{x} > 0$. The Gauss-Seidel iterative method converges
for all symmetric positive definite matrices. In this dissertation, we show that
matrix symmetry can be exploited to improve the performance of the tile growth
algorithms.

2.6 Experimental Results for Locality

To evaluate the possible benefits of sparse tiling transformations, we
compare the performance of Gauss-Seidel for CSR (Figure 2.5) to that of sparse-
tiled Gauss-Seidel (Figure 2.6). We look at the results for both tile growth possi-
sibilities, cache blocking [35] and full sparse tiling.

We ran Gauss-Seidel (GS) for CSR, cache-blocked GS, and full sparse-
tiled GS on five machines, whose characteristics are listed in Table 2.6. As input
<table>
<thead>
<tr>
<th>Matrix</th>
<th>rows</th>
<th>nonzeros</th>
<th>avg nonzeros per row</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Bar</td>
<td>74,926</td>
<td>1,037,676</td>
<td>13.85</td>
</tr>
<tr>
<td>3D Bar</td>
<td>122,061</td>
<td>4,828,779</td>
<td>39.56</td>
</tr>
<tr>
<td>Sphere150K</td>
<td>154,938</td>
<td>11,508,390</td>
<td>74.28</td>
</tr>
<tr>
<td>Pipe</td>
<td>381,120</td>
<td>15,300,288</td>
<td>40.15</td>
</tr>
</tbody>
</table>

Table 2.7: Descriptions of input matrices used in serial experiments

for the experiments described in this chapter, we use the sparse matrices summarized in Table 2.7. Section 2.5 describes the benchmark of matrices used in more detail. The results for these serial experiments are given in terms of normalized execution time, where the execution time of the cache-blocked GS or full sparse-tiled GS excluding the inspector time (“overhead”) is divided by the execution time of the baseline Gauss-Seidel code.

We first describe a simple model for selecting the number of seed partition cells based on a target seed partition size. We then explore the effect of selecting fractions of the L1 and L2 caches as targets on the performance of the sparse-tiled codes. Next, we explore how the number of convergence iterations used for tile growth affects performance. A direct comparison of cache blocking and full sparse tiling performance improvements follow. Finally, we describe the overhead due to the inspector and how that overhead improves by making certain domain-specific assumptions.

### 2.6.1 Partition Size Selection

Both sparse tiling transformations require a seed partitioning of the matrix graph. This section describes a simple model for determining the average partition size based on the memory requirements for the computation within a partition cell. The input parameters are the number of unknowns $R$, the number
Figure 2.16: On the Power3, normalized execution times of executor as the target seed partition size varies.

of nonzeros $NZ$ in the matrix, and the number of convergence iterations $T$.

In Gauss-Seidel, each unknown is iteratively computed, $u_v = (u_v - \sum_{w \neq v} a_{vw} \cdot u_w) / a_{vv}$. When the sparse matrix is stored in a compressed sparse row (CSR) format, the computation for each unknown accesses two doubles for the array entries $u$ and $f$, one integer for an entry in the row index array $ia$, and on average $NZ/R$ doubles for the nonzeros within the associated sparse matrix row, and $NZ/R$ integers for the column identities in the $ja$ array. When updating the unknowns within each partition cell there is an average of $R/k$ unknown computations, where $k$ is the number of seed partitions. The number of bytes needed to store the data associated to one partition cell of computation is given by Equation 2.19, where $si$ is size of an integer in bytes and $sd$ is size of a double.
Figure 2.17: On the UltraSparc-IIIi, normalized execution times of executor as the target seed partition size varies.

in bytes.

\[ B = 2 \frac{R}{k} \cdot sd + (R/k+1) \cdot si + (R/k)(NZ/R) \cdot sd + (R/k) \cdot (NZ/R) \cdot si \]  \hspace{1cm} (2.19)

Solving for \( k \) results in Equation 2.20.

\[ k = \frac{(2 \cdot R \cdot sd + R \cdot si + NZ \cdot (sd + si)) / (B - si)}{(B - si)} \] \hspace{1cm} (2.20)

For each machine, we explore various fractions of the L1 and L2 cache as target seed partition sizes using the simple model. The variety of target seed partition sizes are explored because currently there is no model for how much tile growth will affect the range of tile memory footprints. Also, the simple model does not take into account other uses of the cache (for example, the memory required for the schedule data structure, \texttt{sched}). The performance on the two
Figure 2.18: On Pentium 4 (2GHz), normalized execution times of executor as the target seed partition size varies.

Intel Pentiums and the MIPS R10000 does not significantly differ with the varying target seed partition sizes. Figures 2.16, 2.17, 2.18, and 2.19 show the results for the IBM Power3, Sun UltraSparc-IIi, Intel Pentium 4 (2GHz), and MIPS R10000 respectively (note the logarithmic scale for the target seed partition size). The baseline is two convergence iterations of Gauss-Seidel for CSR with no sparse tiling. The open symbols indicate cache-blocked Gauss-Seidel, and the filled symbols are for full sparse-tiled Gauss-Seidel. The circles represent the results for the 2D bar dataset, and the squares represent the results for the 3D bar dataset.

On the Power3 (Figure 2.16) and the UltraSparc-IIi (Figure 2.17), it is clear that the target seed partition size can affect the performance of sparse-tiled codes. On both these machines targeting the full L1 cache size results in the best
full sparse tiling performance for the 2D bar dataset. For the Power3, targeting the full L1 cache size also exhibits the best results for the 3D bar dataset. The missing L1 cache fractions on the UltraSparc-IIIi for the 3D bar problem set are due to insufficient memory. When targeting L1 cache on these two machines full sparse tiling outperforms cache blocking. This is because the final “clean-up” tile used in cache blocking is larger when more partitions are used in the seed partitioning, and more partitions are needed for smaller target seed partition sizes.

For the Pentium 4 (2GHz), the results shown in Figure 2.18 suggest that target seed partition size does not have a large effect on performance in the range tested. The MIPS R10000 results in Figure 2.19 show a preference toward seed partitions that fit into 1/8 of L2 or the L1 cache.
Figure 2.20: On the Power3, normalized execution times of executor as the number of convergence iterations for the computation and tile growth varies.

2.6.2 Number of Convergence Iterations

One of the parameters for sparse tiling is the number of convergence iterations used for tile growth. Sparse tiling transformations do not provide inter-iteration locality unless two or more convergence iterations are used. If there are more convergence iterations in the computation than the number of convergence iterations used for tile growth, the same sparse-tiled schedule can be used multiple times. In our experiments the number of convergence iterations used for tile growth is the same number of convergence iterations in the computation.

Figure 2.20 shows how the performance of both sparse tiling transformations differ based on the number of convergence iterations on a 2D bar problem (circles) and a 3D bar problem (squares) on the IBM Power3. Notice that each
Figure 2.21: On the UltraSparc-III, normalized execution times of executor as the target seed partition size varies.

Sparse tiling transformation has different behavior on the 2D and 3D problems. This behavior is not consistent with the other machines (see Figures 2.21, 2.22, and 2.23). Therefore, when determining the number of convergence iteration for tile growth the effect of the tile growth method, dataset, and machine must all be taken into account.

Since in some multigrid applications only one or two iterations of Gauss-Seidel are used at one time [2], the rest of our experimental results focus on the case where the number of convergence iterations in the computation and the number of convergence iterations used for tile growth is two, $T = 2$. 
Figure 2.22: On the Pentium 4 (2GHz), normalized execution times of executor as the target seed partition size varies.

2.6.3 Full Sparse-Tiled Gauss-Seidel vs. Cache-Blocked Gauss-Seidel

Table 2.8 provides a side-by-side comparison of the best normalized execution time, chosen from all possible target seed partition sizes, for cache-blocked and full sparse-tiled Gauss-Seidel. Although full sparse-tiled Gauss-Seidel exhibits lower normalized execution times on the majority of machines and problems, the actual difference between a serial execution of full sparse-tiled Gauss-Seidel and cache-blocked Gauss-Seidel is minor.

On the MIPS R10000 we use the PAPI [82] performance analysis package to access the hardware performance counters. Figure 2.24 shows how full sparse-tiled and cache-blocked Gauss-Seidel affect the number of TLB, L2 cache, and L1 cache misses. The effect is different for each problem, thus reinforcing other
Figure 2.23: On the MIPS R10000, normalized execution times of executor as the target seed partition size varies.

results that suggest a performance model must take the characteristics of specific sparse matrices into account.

2.6.4 Overhead

The results in Figures 2.16-2.24 and Table 2.8 all show the normalized execution times obtained by cache-blocked and full sparse-tiled Gauss-Seidel with respect to Gauss-Seidel for CSR without taking the overhead of the inspector into account. It is important to look at the improvements without overhead because Gauss-Seidel can be called multiple times on the same sparse matrix within an algorithm like multigrid. Therefore, when amortized over multiple calls to the computation, an overall improvement is achievable.

Table 2.9 gives the overhead due to the inspector and savings due to the
<table>
<thead>
<tr>
<th>Serial Gauss-Seidel with 2 convergence iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>Power3</td>
</tr>
<tr>
<td>R10000</td>
</tr>
<tr>
<td>Ultra-Sparc-Ii</td>
</tr>
<tr>
<td>Pentium 4 (2GHz)</td>
</tr>
<tr>
<td>Pentium 4 (1.7GHz)</td>
</tr>
</tbody>
</table>

Table 2.8: Comparing the best normalized execution times of cache-blocked Gauss-Seidel and full sparse-tiled Gauss-Seidel across various matrices. Any value less than one indicates a reduction in the execution time. The baseline is Gauss-Seidel for CSR with two convergence iterations. The missing results are due to memory limitations on a UltraSparc-Ii and a Pentium 4 (1.7GHz).

executor in seconds for the Pentium 4 (2GHz) with a 512KB L2 cache. We give the results for five different tile growth algorithms. The first three, our implementation of Cache Blocking [35], FullSparseNaive, and FullSparseWorkSet, only assume that the input sparse matrix is square. The last two, Cache Blocking (Sym) and FullSparseWorkSet (Sym), assume that the sparse matrix is symmetric.

The FullSparseWorkSet algorithm, detailed in Section 2.4.7, uses WorkSet data structures to reduce the amount of work required for tile growth. However, because FullSparseWorkSet must construct a symmetric matrix graph if one does not already exist, only FullSparseWorkSet (Sym) and not FullSparseWorkSet results in less overhead than FullSparseNaive.

The number of calls to the sparse-tiled Gauss-Seidel with 2 convergence iterations needed to amortize the overhead is the break even point and is calculated as \( \text{OH/savings} \) rounded up to the nearest integer. For example, on the 2D bar mesh, the overhead of using FullSparseWorkSet (Sym) requires that
Figure 2.24: Shows how cache-blocked and full sparse-tiled Gauss-Seidel affect the number of TLB, L2 cache, and L1 cache misses on the MIPS R10000.

the rescheduled Gauss-Seidel be executed 51 times for an overall speedup (see Table 2.9). All of the overhead times are of this magnitude or higher. Profiling the sparse tiling inspectors indicates that the Metis library partitioner accounts for the majority of the overhead time. Since Metis was created to partition irregular meshes as a preprocessing step, more efficient graph partitioning heuristics should be used with run-time sparse tiling inspectors.

For problems that occur in Finite Element Analysis applications, rows in the sparse matrix have identical nonzero structure if they are associated with the same node in a mesh. In [35], the authors do not partition the matrix graph, but instead partition the original mesh. This dramatically reduces the overhead because the number of nonzeros needed to express a mesh when there are two unknowns per mesh node is \(NZ/4\) where \(NZ\) is the number of nonzeros in the
Sparse tiling the resulting sparse matrix
Gauss-Seidel kernel with 2 convergence iterations on Pentium 4 (2GHz)

<table>
<thead>
<tr>
<th>Tile Growth Algorithm</th>
<th>2D bar</th>
<th></th>
<th>3D bar</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OH</td>
<td>savings</td>
<td>BE</td>
<td>OH</td>
</tr>
<tr>
<td>Cache Blocking</td>
<td>0.860</td>
<td>0.013</td>
<td>67</td>
<td>6.649</td>
</tr>
<tr>
<td>FullSparseNaive</td>
<td>0.777</td>
<td>0.013</td>
<td>60</td>
<td>6.647</td>
</tr>
<tr>
<td>FullSparseWorkSet</td>
<td>0.849</td>
<td>0.013</td>
<td>66</td>
<td>6.821</td>
</tr>
<tr>
<td>Cache Blocking (Sym)</td>
<td>0.669</td>
<td>0.013</td>
<td>52</td>
<td>4.829</td>
</tr>
<tr>
<td>FullSparseWorkSet (Sym)</td>
<td>0.660</td>
<td>0.013</td>
<td>51</td>
<td>5.067</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tile Growth Algorithm</th>
<th>Sphere</th>
<th></th>
<th>Pipe</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OH</td>
<td>savings</td>
<td>BE</td>
<td>OH</td>
</tr>
<tr>
<td>Cache Blocking</td>
<td>9.724</td>
<td>-0.003</td>
<td>-</td>
<td>13.497</td>
</tr>
<tr>
<td>FullSparseNaive</td>
<td>11.474</td>
<td>-0.005</td>
<td>-</td>
<td>12.704</td>
</tr>
<tr>
<td>FullSparseWorkSet</td>
<td>10.597</td>
<td>-0.005</td>
<td>-</td>
<td>13.621</td>
</tr>
<tr>
<td>Cache Blocking (Sym)</td>
<td>8.730</td>
<td>-0.002</td>
<td>-</td>
<td>11.418</td>
</tr>
<tr>
<td>FullSparseWorkSet (Sym)</td>
<td>9.735</td>
<td>-0.005</td>
<td>-</td>
<td>11.541</td>
</tr>
</tbody>
</table>

Table 2.9: Gives the overhead due to the inspector (OH) and savings incurred by the executor in seconds on the Pentium 4 (2GHz) for two iterations of Gauss-Seidel. The overhead divided by the savings and rounded up is the break even (BE) point, or the number of Gauss-Seidel calls needed to amortize the overhead. We give the results for five different tile growth algorithms. The first three, our implementation of Cache Blocking, FullSparseNaive, and FullSparseWorkSet, only assume that the input sparse matrix is square. The last two, Cache Blocking (Sym) and FullSparseWorkSet (Sym), assume that the sparse matrix is symmetric.

resulting sparse matrix. For a 3D mesh the number of nonzeros is \(NZ/9\). The number of nodes in the mesh is \(R/2\) for a 2D mesh and \(R/3\) for a 3D mesh.

Table 2.10 shows the new overhead and savings results when the original mesh is sparse tiled versus the resulting sparse matrix. The savings in seconds are almost identical to those in Table 2.9 but the overhead decreases significantly. Similar results occur on the other machines as well. Since the Sphere dataset was not generated by the FETk package, we did not recreate its original mesh.
Sparse tiling the original mesh
Gauss-Seidel with 2 convergence iterations on Pentium 4 (2GHz)

<table>
<thead>
<tr>
<th>Tile Growth Algorithm</th>
<th>2D bar</th>
<th></th>
<th>3D bar</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OH</td>
<td>savings</td>
<td>BE</td>
<td>OH</td>
</tr>
<tr>
<td>FullSparseWorkSet</td>
<td>0.367</td>
<td>0.012</td>
<td>31</td>
<td>1.677</td>
</tr>
<tr>
<td>Cache Blocking</td>
<td>0.372</td>
<td>0.012</td>
<td>31</td>
<td>1.651</td>
</tr>
<tr>
<td>FullSparseWorkSet (Sym)</td>
<td>0.344</td>
<td>0.012</td>
<td>29</td>
<td>1.540</td>
</tr>
<tr>
<td>Cache Blocking (Sym)</td>
<td>0.351</td>
<td>0.012</td>
<td>30</td>
<td>1.486</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tile Growth Algorithm</td>
<td>Sphere</td>
<td></td>
<td>Pipe</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OH</td>
<td>savings</td>
<td>BE</td>
<td>OH</td>
</tr>
<tr>
<td>FullSparseWorkSet</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4.166</td>
</tr>
<tr>
<td>Cache Blocking</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4.111</td>
</tr>
<tr>
<td>FullSparseWorkSet (Sym)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3.983</td>
</tr>
<tr>
<td>Cache Blocking (Sym)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3.936</td>
</tr>
</tbody>
</table>

Table 2.10: Shows the new overhead (OH) and savings results when the original mesh is sparse tiled versus the sparse matrix resulting from FEA. The overhead divided by the savings and rounded up is the break even (BE) point, or the number of Gauss-Seidel calls needed to amortize the overhead. Since the Sphere matrix was not generated by the FEtk package, we did not recreate its original mesh.

### 2.7 Sparse Tiling other Stationary Iterative Methods

Other stationary iterative methods include Successive Overrelaxation (SOR) and Jacobi [14]. In general, stationary iterative methods update the unknown vector \( \tilde{u} \) for some number of convergence iterations, and use the values of related unknowns while updating each unknown \( u_i \). Since the number of convergence iteration \( T \) and the matrix graph, or relationship between the unknowns, are input for Gauss-Seidel sparse tiling inspectors, these same inspectors can be used to sparse tile the other stationary iterative methods with little or no modifications.

The SOR method exhibits the same intra-iteration and inter-iteration data dependences as Gauss-Seidel. Therefore, the `FULLSPARSENAIVE_GSCSR`
Algorithm `FullSparseNaive_JacobiCSR(G(V,E),part(),T,iter_s)`

1: foreach vertex $v \in V$, $\theta(iter_s, v) \leftarrow part(v)$

**Downward tile growth**
2: for $iter = (iter_s - 1)$ downto 1
3:     foreach vertex $v \in V$, $\theta(iter, v) \leftarrow \theta(iter + 1, v)$
4:     foreach $<v, w> \in E$
5:         $\theta(iter, v) \leftarrow \min(\theta(iter, v), \theta(iter + 1, w))$
6:         $\theta(iter, w) \leftarrow \min(\theta(iter, w), \theta(iter + 1, v))$
7:     end foreach
8: end for

**Upward tile growth**
9: for $iter = (iter_s + 1)$ to $T$
10:    foreach vertex $v \in V$, $\theta(iter, v) \leftarrow \theta(iter - 1, v)$
11:    foreach $<v, w> \in E$
12:        $\theta(iter, v) \leftarrow \max(\theta(iter, v), \theta(iter - 1, w))$
13:        $\theta(iter, v) \leftarrow \max(\theta(iter, w), \theta(iter - 1, v))$
14:    end foreach
15: end for

Figure 2.25: `FullSparseNaive_JacobiCSR` Algorithm

and `FullSparseWorkSet_GSCSR` algorithms can be used to grow sparse tiles throughout the SOR iteration space with no modifications. This is done in Chapter 4 as part of an extension to the PETSc software package.

In the Jacobi method, the unknown values computed in the previous convergence iteration are stored separately from the unknowns being computed for the current convergence iteration. While updating an unknown within the current convergence iteration, only the related unknown values from the previous convergence iteration are used. Therefore, Jacobi iteration spaces have no intra-iteration dependences. Although a sparse tiling generated for Gauss-Seidel can legally be used for Jacobi, Jacobi has fewer constraints and a simpler inspector can be used. Figure 2.25 shows how the `FullSparseNaive_GSCSR` algorithm
can be simplified to full sparse tile a Jacobi iteration space. The reordering function $\sigma$ has no constraints and can be the identity function. Although this is the case, for improved intra-iteration locality, a consecutive ordering of the unknowns based on their seed partitioning assignments should result in better intra-iteration locality.

2.8 Related Work

Work related to performance transformations for stationary iterative methods can be categorized by whether it deals with regular or irregular meshes and whether it attempts to improve intra-iteration locality and/or inter-iteration locality. Another important distinction is between code transformations that have been automated in a compiler versus programming transformations that require some domain-specific knowledge and are currently applied by hand.

Traditional tiling [132, 66, 41, 131, 129, 21, 86] can be applied to a perfect loop nest that traverses the unknowns associated with a regular mesh, provided the memory references and loop boundaries are affine and the unknowns are ordered in a way that allows a compile-time analysis to find a legal tiling. In particular, after the enabling transformation skewing is applied, tiling is often applicable to Gauss-Seidel and SOR over a regular mesh.

For Jacobi over a regular mesh, tiling transformations developed for imperfectly nested loops can be used [117, 6]. Another issue involved in tiling computations on regular meshes is how to determine the tiling and array padding parameters [107]. If other compiler transformations, such as skewing, function inlining and converting ‘while’ loops to ‘for’ loops, are used, then it is possible to apply tiling transformations for imperfectly nested loops to stationary iterative methods to achieve inter-iteration locality on regular meshes.

Increasing inter-iteration locality through programming transformations
for iterative stationary methods on regular meshes is explored by [35], [114], [15], [68], [134], and [43]. Such programming transformations are explored extensively due to the prevalence of iterative regular mesh computations that compilers do not tile because the necessary combination of enabling transformations are not done automatically.

Both iteration space slicing [98] and data shackling [76] are transformations that divide up the iteration space based on an initial data partition. In this manner, they are able to improve intra-loop and inter-loop data locality. This is similar to what the sparse tiling transformation 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.

There has also been work on compiler-generated inspectors/executors for improving intra-iteration locality of irregular problems [91, 33, 51, 88]. These papers describe how a compiler can analyze non-affine array references in a loop and generate the inspectors and executors for performing run-time data and iteration reordering. These transformations 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 [64, 65] have developed a code generator, SPARSITY, that improves the intra-iteration locality for the $\bar{x}$ and $\bar{b}$ vectors in the sparse matrix-vector multiplication $A\bar{x} = \bar{b}$. Their work does not address the issue of improving inter-iteration locality for the multiple convergence iterations of stationary methods.

The only other technique to our knowledge that handles inter-iteration locality for irregular meshes is unstructured cache blocking [35]. Cache blocking and full sparse tiling are the programming transformations that this dissertation classifies as sparse tiling transformations.
2.9 Summary

This chapter provides the inspector/executor framework for implementing the sparse tiling transformations cache blocking for irregular grids [35] and full sparse tiling [119]. Cache blocking and full sparse tiling improve the intra-iteration and inter-iteration data locality for the Gauss-Seidel computation. Based on an analysis of the data dependences in the executor, we prove that full sparse-tiled Gauss-Seidel generates bit-equivalent results to traditional Gauss-Seidel that uses the same data order.

Our serial experimental results show that the sparse-tiled Gauss-Seidel executor exhibits reduced execution time on five different machines. We give a simple model for selecting the number of cells in the seed partitioning based on the amount of memory each partition cell requires. The most effective target seed partition size depends on the machine, the input sparse matrix, the number of convergence iterations, and the tile growth technique (cache blocking or full sparse tiling).

Although in the majority of cases, full sparse-tiled Gauss-Seidel performs better than cache-blocked Gauss-Seidel, the performance of the two transformations is similar in serial experiments. Amortizing the overhead for either of the sparse tiling transformations currently requires between 36-242 calls to Gauss-Seidel with two convergence iterations on our benchmark matrices. This is mostly due to the graph partitioning heuristic used. By performing the sparse tiling transformations on the original mesh and assuming the mesh is symmetric in structure the range of overheads becomes between 9-70 calls to Gauss-Seidel with two convergence iterations.

The text of this chapter is in part a reprint of the material as it appears in the proceedings of the 2001 International Conference on Computational Science (ICCS) and a technical report, and as it will appear in the International Journal
of High Performance Computing Applications. The dissertation author was the primary researcher and author for these publications, which form the basis for this chapter.
Chapter 3

Sparse Tiling Stationary Iterative Methods for Parallelism

Sparse tiling transformations can be used to parallelize stationary iterative methods as well as improving the intra-iteration and inter-iteration locality. This chapter describes how full sparse-tiled Gauss-Seidel can be parallelized and presents experimental results comparing both sparse tiling transformations on shared memory processors. Section 3.1 details the extensions to the sparse tiling inspector and executor needed to parallelize the sparse tiling executor. In Section 3.2, parallel full sparse-tiled Gauss-Seidel is compared with other Gauss-Seidel parallelization techniques.

3.1 Executing Sparse-Tiled Gauss-Seidel in Parallel

The sparse tiling inspector performs a run-time data and iteration reordering by partitioning the matrix graph, growing tiles from the cells of the seed partitioning, constructing the data reordering function, and creating the new schedule based on the tiling function. In order to execute sparse tiles in parallel we construct a *tile dependence graph*, which indicates the dependences between
tiles. The tile dependence graph is executed by a master-worker implementation. The master maintains the tile dependence graph and puts tiles whose data dependences are satisfied on a ready queue. The workers dequeue tiles from the ready queue, execute the tiles, and notify the master upon tile completion.

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

- **Partition** the matrix graph to create the seed partitioning. Each piece of the partition is called a *cell*.

- **Choose a numbering** for the cells of the seed partition.

- **Grow tiles** from each cell of the seed partitioning in turn to create the tiling function, which assigns each iteration point to a tile.

- **Generate** the data reordering function

- **Remap** the data using the data reordering function.

- **Reschedule** by creating a schedule function, based on the tiling function.

- **Construct a tile dependence graph**, thus identifying tiles that can be executed in parallel.

As with sparse tilings intended for serial executions, either cache blocking tile growth [35] or full sparse tile growth (called serial sparse tiling in [119]) can be used to grow tiles based on an initial matrix graph partitioning. We implement and show results for both methods. Note that scheduling for parallelism adds two steps to the sparse tiling inspector. Now the inspector must more carefully select a numbering for the cells in the partitioning, and the inspector must construct a tile dependence graph.
3.1.1 Increasing Parallelism with Graph Coloring

This section describes the importance of the numbering given to the seed partitioning. The degree of parallelism within sparse-tiled Gauss-Seidel executor is a function of the tile dependence graph. Specifically, 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 3.1 gives two possible sparse tilings of the same two-dimensional iteration space. The tile dependence graphs for both sparse tilings are shown in Figure 3.2. The first tile dependence graph, which exhibits no parallelism, has height equal to six and average parallelism equal to one. The second tile dependence graph has height three and average parallelism two. 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 how the cells of the seed partitioning are numbered. The tile growth algorithms use the numbering to indicate the execution order for adjacent tiles, thus the partition numbering affects the data dependence direction between tiles.

We compare the partition numbering assigned 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 $(A, B)$ in the partition graph. We color the nodes of the partition graph and assign consecutive numbers to the cells of the partitioning that corre-
Figure 3.1: Two applications of sparse tiling to a two-dimensional iteration space. In both cases, the first row of iterations is partitioned into six cells and then tiles are grown upward. In the top diagram, the seed partitions are ordered from left to right. The small arrows indicate dependences between tiles.

Figure 3.2: Tile dependence graphs for the sparse tilings shown in Figure 3.1. 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.
Figure 3.3: The average parallelism in the tile dependence graph for full sparse-tiled Gauss-Seidel with 2 convergence iterations. The target seed partition size is 1/8 of L2 cache.

This way, the tiles grown from cells of a given color will not be adjacent and therefore probably not be data dependent. In our experiments, we use the greedy heuristic provided in the Graph Coloring Programs [27] to color the partition graph. After tile growth, the data dependences between tiles must still be calculated to insure correctness (as will be described in Section 3.1.2), since even though two partition cells are not adjacent, the tiles grown from the cells may be dependent.

The graph in Figure 3.3 shows the average parallelism for four different matrices with full sparse tile growth 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.
Figure 3.4: The effect that graph coloring has on speedup for the parallel full sparse-tiled Gauss-Seidel executor performing 2 convergence iterations. These speedups are from experiments run on one shared memory node of the Blue Horizon at SDSC. Each node has 8 Power3 processors.

The importance of the average parallelism in the tile dependence graph can be seen when we examine the speedup of parallel full sparse-tiled Gauss-Seidel executor using the three different partition numberings. In Figure 3.4, notice that the top two lines showing speedups for the Pipe matrix indicate 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 3.4.
3.1.2 Constructing the Tile Dependence Graph

This section details the construction of the tile dependence graph. Since the tile dependence graph is constructed at runtime by the sparse tiling inspector, it is important that the algorithm is efficient. In fact, it is possible to leverage the work done in the tile growth algorithm (see Figure 2.8). The CONSTRUCTTDG algorithm shown in Figure 3.5, uses the \textit{NodeOrd} relation that the tile growth algorithm generates. This reduces the amount of work required, because otherwise, the CONSTRUCTTDG algorithm would have to traverse all the dependences in the Gauss-Seidel iteration space to determine which ones cut across tile boundaries and therefore caused a tile dependence.

The following lemmas are used to show the correctness of the tile dependence graph generated in the CONSTRUCTTDG algorithm. Intuitively, if there is a dependence between two iteration points in the iteration space, and those two iterations are assigned to \textit{tile}_1 and \textit{tile}_2 respectively, then there is a dependence from \textit{tile}_1 to \textit{tile}_2 which we will denote with \textit{tile}_1 \delta \textit{tile}_2.

\textbf{Lemma 10} Upon completion of the CONSTRUCTTDG algorithm, if \( \exists \) iter, \( v \) such that \( 1 \leq \text{iter} \leq (T - 1) \) and \( v \in V \) and \( \theta(\text{iter}, v) < \theta(\text{iter} + 1, v) \), then \( \theta(\text{iter}, v) \delta \theta(\text{iter} + 1, v) \).

\textbf{Lemma 11} Upon completion of the CONSTRUCTTDG algorithm, if \( \exists \) iter, \( v, w \) such that \( 1 \leq \text{iter} \leq T \) and \( < v, w > \in \text{NodeOrd} \) and \( \theta(\text{iter}, v) < \theta(\text{iter}, w) \), then \( \theta(\text{iter}, v) \delta \theta(\text{iter}, w) \).

\textbf{Lemma 12} Upon completion of the CONSTRUCTTDG algorithm, if \( \exists \) iter, \( v, w \) such that \( 1 \leq \text{iter} \leq (T - 1) \) and \( < v, w > \in \text{NodeOrd} \) and \( \theta(\text{iter}, w) < \theta(\text{iter} + 1, v) \), then \( \theta(\text{iter}, w) \delta \theta(\text{iter} + 1, v) \).

\textbf{Proof} for Lemmas 10, 11, and 12: Each of these lemmas follow directly from the pseudocode in Figure 3.5. \qed
Algorithm ConstructTDG(V, θ, NodeOrd, T)

1: \( TDG \leftarrow \emptyset \)
2: foreach vertex \( v \in V \)
3: \hspace{1em} for iter = 1 to \( T - 1 \)
4: \hspace{2em} if \( \theta(\text{iter}, v) < \theta(\text{iter} + 1, v) \) then
5: \hspace{3em} \( TDG \leftarrow TDG \cup \{\theta(\text{iter}, v) \\delta \theta(\text{iter} + 1, v)\} \)
6: \hspace{1em} end for
7: \hspace{1em} end foreach
8: foreach \( <v, w> \in \text{NodeOrd} \)
9: \hspace{1em} for iter = 1 to \( T - 1 \)
10: \hspace{2em} if \( \theta(\text{iter}, v) < \theta(\text{iter}, w) \) then
11: \hspace{3em} \( TDG \leftarrow TDG \cup \{\theta(\text{iter}, v) \\delta \theta(\text{iter}, w)\} \)
12: \hspace{1em} end for
13: \hspace{1em} end foreach
14: \hspace{1em} if \( \theta(T, v) < \theta(T, w) \) then
15: \hspace{2em} \( TDG \leftarrow TDG \cup \{\theta(T, v) \\delta \theta(T, w)\} \)
16: \hspace{1em} end foreach

Figure 3.5: ConstructTDG Algorithm

In the proof of correctness for the full sparse tiling inspector (see Section 2.4.6), the constraints on the tiling function \( \theta \) and reordering function \( \sigma \) were rewritten in terms of the matrix graph representation. Those same constraints specify when a dependence exists between tiles.

**Theorem 3** Let \( G(V, E) \) be the directed matrix graph for a square sparse matrix \( A \). Let \( \theta \) and \( \sigma \) functions be those generated by the full sparse tiling inspector using FullSparseNaive_GSCSR as the tile growth algorithm. The tile dependence graph relation \( TDG \) generated in the ConstructTDG algorithm includes a tile dependence either directly or indirectly through transitivity if and only if one or
more of the following conditions holds:

1. \( \exists v, \text{iter}_1, \text{iter}_2 \text{ such that } v \in V \text{ and } (1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \text{ and } \theta(\text{iter}_1, v) < \theta(\text{iter}_2, v) \)

2. \( \exists v, w, \text{iter} \text{ such that } v, w \in V \text{ and } \sigma(v) < \sigma(w) \text{ and } (<v, w> \in E \text{ or } <w, v> \in E) \text{ and } (1 \leq \text{iter} \leq T) \text{ and } \theta(\text{iter}, v) < \theta(\text{iter}, w) \)

3. \( \exists v, w, \text{iter}_1, \text{iter}_2 \text{ such that } v, w \in V \text{ and } \sigma(v) \neq \sigma(w) \text{ and } (<v, w> \in E \text{ or } <w, v> \in E) \text{ and } (1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \text{ and } \theta(\text{iter}_1, v) < \theta(\text{iter}_2, w) \)

**Proof:**

First we show that if the CONSTRUCTTDG algorithm includes a dependence in the relation \( TDG \), then Conditions 1, 2, or 3 hold. Lemmas 10, 11, and 12 specify under what conditions the CONSTRUCTTDG algorithm includes a dependence in the relation \( TDG \). For the dependences included into \( TDG \) due to Lemma 10, Condition 1 holds. For the dependences included into \( TDG \) due to Lemma 11, Condition 2 holds due to Lemma 2. For the dependences included into \( TDG \) due to Lemma 12, Condition 3 holds since Lemma 2 requires that for any \(<v, w> \in \text{NodeOrd}\) there must be an edge in the matrix graph.

At this point, we have shown that all of the tile dependences generated by the CONSTRUCTTDG algorithm satisfy at least one of the conditions that indicate an actual tile dependence exists. We next show the other direction; if any of the dependence conditions exist then the CONSTRUCTTDG algorithm includes those dependences either directly or indirectly through transitivity in the tile dependence graph.

A tile dependence due to Condition 1 will be included in the tile dependence graph either directly or transitively due to Lemma 10. A tile dependence due to Condition 2 will be included due to Lemma 11. A tile dependence due to Condition 3 will be included transitively either due to a combination of Lemma 10
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>L1 cache</th>
<th>L2 cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra</td>
<td>SUN HPC10000, up to 32 400 MHz UltraSPARCII processors cc -fast -DNDEBUG</td>
<td>16KB</td>
<td>4MB</td>
</tr>
<tr>
<td>Blue Horizon Node</td>
<td>One node of an IBM SP, Eight 375 MHz Power3 processors xlc -bmaxdata:0x80000000 -bmaxstack:0x10000000 -O3 -DNDEBUG</td>
<td>64KB</td>
<td>8MB</td>
</tr>
</tbody>
</table>

Table 3.1: Descriptions of computer architectures used in parallel experiments

<table>
<thead>
<tr>
<th>Matrix</th>
<th>rows</th>
<th>nonzeros</th>
<th>avg nonzeros per row</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Bar</td>
<td>74,926</td>
<td>1,037,676</td>
<td>13.85</td>
</tr>
<tr>
<td>3D Bar</td>
<td>122,061</td>
<td>4,828,779</td>
<td>39.56</td>
</tr>
<tr>
<td>Sphere150K</td>
<td>154,938</td>
<td>11,508,390</td>
<td>74.28</td>
</tr>
<tr>
<td>Pipe</td>
<td>381,120</td>
<td>15,300,288</td>
<td>40.15</td>
</tr>
<tr>
<td>Wing903K</td>
<td>924,672</td>
<td>38,360,266</td>
<td>41.49</td>
</tr>
</tbody>
</table>

Table 3.2: Descriptions of input matrices used in parallel experiments

and Lemma 11 or due to a combination of Lemma 10 and Lemma 12.

### 3.1.3 Parallel Executor Speedup

Our parallel experiments are conducted using the IBM Blue Horizon and SUN Ultra at the San Diego Supercomputer center. Details on both machines are given in Table 3.1. Table 3.2 summarizes the matrices used in these experiments. For more detail on the matrix benchmarks see Section 2.5.

Our experiments examine the speedup of parallel sparse-tiled Gauss-Seidel executor over a typical serial Gauss-Seidel (as shown in Figure 2.5), the overhead due to the sparse tiling inspector, and the average parallelism within the tile dependence graph.

Figure 3.6 shows the speedups for parallel cache-blocked and full sparse-
### Table 3.3: The number of Gauss-Seidel executions required to amortize sparse tiling inspector overhead. Each Gauss-Seidel execution performs two convergence iterations.

<table>
<thead>
<tr>
<th>Input Matrix</th>
<th>Overhead (sec)</th>
<th>Savings/Execution (sec)</th>
<th>Break Even (# of executions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>n=2</td>
<td>n=4</td>
</tr>
<tr>
<td>Matrix9</td>
<td>2.03</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>Matrix12</td>
<td>13.69</td>
<td>0.14</td>
<td>0.20</td>
</tr>
<tr>
<td>Sphere150K</td>
<td>31.41</td>
<td>0.17</td>
<td>0.26</td>
</tr>
<tr>
<td>PipeOT15mill</td>
<td>48.28</td>
<td>0.39</td>
<td>0.52</td>
</tr>
<tr>
<td>Wing903K</td>
<td>116.86</td>
<td>0.65</td>
<td>0.96</td>
</tr>
</tbody>
</table>

Tiled Gauss-Seidel executing on 2, 4, and 8 processors of a IBM Blue Horizon node. Figure 3.7 shows the parallel executor speedups on the SUN Ultra using up to 32 processors. In [35], cache blocking uses seed partitions that fit into half of L2 cache; this parameter choice is duplicated in our implementation of cache blocking and typically results in the best performance for cache blocking. In our experience, full sparse tiling Gauss-Seidel executors experience better performance when the seed partition size fits into one-eighth of L2 cache.

While both sparse tiling transformations achieve speedups over the un-optimized serial Gauss-Seidel, full sparse tiling often achieves the best speedups. With cache blocking all of the tiles are executed in parallel except for the last tile. This last tile cannot start executing until all the other tiles have completed, so parallelism is inhibited. Further, the last tile can be large and therefore have poor inter-iteration locality and intra-iteration locality.
Figure 3.6: Speedups for the parallel sparse-tiled Gauss-Seidel executor performing 2 convergence iterations over a serial Gauss-Seidel schedule using the typical schedule. These experiments were run on one node of the IBM Blue Horizon at SDSC. Each node has 8 shared memory Power3 processors.

3.1.4 Overhead for Generating a Parallel Schedule

Since the sparse tiling inspector executes at runtime, its overhead must be amortized over all uses of the generated schedule and remapped data. We present the inspector overhead separately because Gauss-Seidel is typically called many times within applications like multigrid. We can amortize the overhead over multiple calls involving the same sparse matrix. Our parallel 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 3.3.

On average, 75% of the overhead is due to the graph partitioner Metis. A break down of the overhead per input matrix is given in Table 3.4. Owner-
Figure 3.7: Speedups for the parallel sparse-tiled Gauss-Seidel executor performing 2 convergence iterations over a serial Gauss-Seidel schedule using the typical schedule. These experiments were run on a SUN HPC10000 which has 36 shared memory UltraSPARCIII processors.

computes parallelization methods [57] for sparse matrices also require a partitioner and data remapping 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, by using the original mesh. The results in this section use the function Metis_PartGraphRecursive() for the matrix graph partitioning. In Section 4.2.4, we show some serial results using the Gpart partitioner [50] to create the seed partitioning instead of Metis. Gpart is more efficient and therefore reduces the inspector overhead.

Previous sparse tiling work [35, 119] performed cache blocking and full 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,
Blue Horizon, Gauss-Seidel with 2 convergence iterations, Reordering for parallelism

<table>
<thead>
<tr>
<th>Input Matrix</th>
<th>Partition Time</th>
<th>Data Remapping</th>
<th>Tile Growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix9</td>
<td>78.92%</td>
<td>14.06%</td>
<td>7.02%</td>
</tr>
<tr>
<td>Matrix12</td>
<td>71.89%</td>
<td>13.42%</td>
<td>14.69%</td>
</tr>
<tr>
<td>Sphere150K</td>
<td>67.64%</td>
<td>16.53%</td>
<td>15.83%</td>
</tr>
<tr>
<td>PipeOT15mill</td>
<td>81.42%</td>
<td>9.73%</td>
<td>8.85%</td>
</tr>
<tr>
<td>Wing903K</td>
<td>83.58%</td>
<td>9.95%</td>
<td>6.47%</td>
</tr>
</tbody>
</table>

Table 3.4: Percentage of the inspector due to generating the seed partitioning, remapping the data, and growing tiles. Around 80% to 90% of the overhead is due to partitioning the matrix graph plus remapping the unknown vector, right-hand side vector, and sparse matrix.

the resulting matrix graph has multiple rows with the same nonzero 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.

### 3.2 Comparison with other Gauss-Seidel Parallelization Transformations

Sparse tiling transformations differ from other Gauss-Seidel parallelization techniques, specifically multi-coloring and owner-computes methods [57], in their focus on improving both intra- and inter-iteration locality. Since in all these parallelization methods each processor is given an approximately equal 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 transformations.

Multi-coloring is the standard way to parallelize irregular Gauss-Seidel [14]. 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 itera-
<table>
<thead>
<tr>
<th>Method</th>
<th>Parallel Efficiency</th>
<th>Intra-iteration locality</th>
<th>Inter-iteration locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-coloring</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Owner-computes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Sparse tiling</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3.5: Summary of how the various Gauss-Seidel parallelization transformations compare in how they handle the three performance aspects

... 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 transformations 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.

3.2.2 Intra-iteration Locality

Multi-coloring transformations have poor intra-iteration locality because in order for iteration point \(<iter, v>\) to be executed in parallel with other iteration points, \(<iter, v>\) 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 generating a reordering for the unknowns based on that partitioning [50].

With sparse tiling transformations, the partitions used to grow tiles are selected so that they fit into (some level of) cache. Therefore the data reordering will result in intra-iteration locality.

3.2.3 Inter-iteration Locality

Both multi-colored 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 transformations subdivide the iteration space so that multiple convergence iterations over a subset of the unknowns occur atomically, thus improving the inter-iteration locality.

### 3.2.4 Experimental Comparison

Since owner-computes methods differ from sparse tiling methods only by their lack of inter-iteration locality, we compare the two experimentally. Our owner-computes implementation is idealized because the Gauss-Seidel dependences are violated between subpartitions in order to give the owner-computes method perfect intra-iteration parallel efficiency. This approximates the performance of a complete Nodal Gauss-Seidel implementation, which has good intra-iteration parallel efficiency. Inter-iteration parallel efficiency within the idealized owner-computes is achieved by giving each processor the same number of unknowns. Finally, intra-iteration locality is provided by partitioning the submatrix graph on each processor and then reordering the unknowns accordingly.

The Sphere, Pipe, and Wing problems are data sets that do not fit into L2 cache once the data is partitioned for parallelism. The Sphere matrix has an average number of 74.28 nonzeros per row (as shown in Table 3.2). This causes sparse tiles to grow rapidly and therefore results in poor parallelism in the tile dependence graph. Recall in Figure 3.3 that the maximum average parallelism was 7.5 for 2 convergence iterations when the full sparse tiling inspector tiles for the L2 cache on the Power3 processor. 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 3.8). However, with the Pipe and Wing matrices the average number of nonzeros per row is much lower at 40.15 and 41.49. Correspondingly, the average parallelism when tiling for one-eighth of the L2 cache on the Power3 processor is...
Figure 3.8: Full sparse-tiled Gauss-Seidel with 2 and 4 convergence iterations compared with the idealized owner-computes computation. These experiments were run on one node of the IBM Blue Horizon at SDSC.

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 Sun Ultra (Figure 3.9), the L2 cache is smaller, so smaller tiles are necessary to fit into one-eighth of the L2 cache. This increases the number of tiles and 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 idealized owner-computes outperforms full sparse tiling in this instance, because in this case full sparse tiling doesn’t generate enough parallelism.

The Wing results on the Ultra are problematic. The tile dependence
Figure 3.9: Full sparse-tiled Gauss-Seidel with 2 and 3 convergence iterations compared with the idealized owner-computes computation. These experiments were run on the SUN HPC10000 at SDSC.

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 out performs owner-computes methods on shared memory architectures. Our idealized owner-computes implementation assumes perfect intra-iteration parallel efficiency and good intra-iteration locality. These results show that inter-iteration locality is an important performance aspect that owner-computes methods miss.
3.3 Related Work

Since the smoother dominates the computation time in multigrid methods, much work revolves around parallelizing the smoother. Gauss-Seidel has good convergence properties; therefore, Adams has developed a parallelization method that parallelizes within a convergence iteration [2]. We focus on parallelizing between the tiles that result from full sparse tiling Gauss-Seidel. Another approach is to use smoothers that are easily parallelized like domain decomposition [116], blocked Jacobi, or blocked Gauss-Seidel [55]. 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 [49].

In the data parallel programming model, the general approach is to specify a data distribution and then have the compiler combined with the run-time library handle the distribution of computation and insertion of communication. For distributed parallelism, the user can specify the data distribution for arrays in programming languages like Fortran D [38, 127], High Performance Fortran (HPF) [56], ZPL [23]. If the loop operates on a regular grid, then the communication patterns needed between distributed processors can be determined at compile time. If the loop operates on an irregular grid, or a regular grid topology that is not specified until runtime, then inspector/executor strategies [90] can be used. The compiler can insert calls to run-time libraries of inspectors and executors that focus on determining the communication pattern and optimizing communication [4, 31, 79]. Specific examples of using such run-time libraries for parallelism within irregular code are described in [30, 63]. There are also run-time libraries that allow users to specify data distribution changes and grid adaptation. For example, KeLP [10] allows the expression of data motion and data decomposition through the use of geometric meta data.
The main difference between full sparse tiling parallelization from regular and irregular data parallel programming, is that data parallel programming focuses on a distribution of the data. Full sparse tiling employs a decomposition of the iteration space as the basis for parallelization. By parallelizing between tiles, full sparse tiling is able to achieve inter-iteration locality as well as parallelism.

When parallelism cannot be determined at compile time, parallelism can be achieved by verifying parallelism through run-time tests, using an inspector/executor framework to develop a run-time schedule, and through speculative parallelization. Compile-time data dependence analysis can generate run-time tests for data dependences [97, 59, 93], and then select between multiple versions of the loop [20], for example a parallel and serial version. Testing the values of variables at runtime is the most efficient run-time test for determining loop parallelism. In some cases where more than constant time checks are required, it is possible to prove independence by doing run-time analysis of the values in the index array [81, 47]. Hybrid Analysis handles the range of run-time test between checking some constants through iterating over all the data dependences [109]. General techniques for run-time parallelization, including speculative parallelization [105] are summarized and evaluated by Rauchwerger in [102]. The goal of run-time parallelization techniques is to maximize the parallelism in a partially parallel loop\(^1\). Sparse tiling transformations generate parallelism between tiles, therefore the parallelism is coarser-grained and result in improved data locality. An open question is to determine which approach is more beneficial to performance.

Another way to handle intra-iteration locality, inter-iteration locality, and parallelism in stationary iterative methods is through the use of overlapping tiles. In the context of regular codes, the idea of having tiles with overlapping computation has been looked at by Su and Carter [43] and within the Cactus

\(^1\)Partially parallel loops require synchronization for legal parallelization
project [8]. With overlapping tiles, different processors do a certain amount of
duplicate computation so as to avoid communication and synchronization costs.
It is possible to generate overlapping tiles for stationary iterative methods on
irregular grids, and the evaluation of such a transformation is possible future
work.

3.4 Summary

Full sparse tiling explicitly creates intra-iteration locality, inter-iteration
locality, and parallelism for irregular Gauss-Seidel. Our parallel experimental
results show that the combination of these three performance aspects results
in improved performance. In shared memory parallel experiments, full sparse-
tiled Gauss-Seidel outperforms cache-blocked Gauss-Seidel. Also, full sparse-tiled
Gauss-Seidel is compared with an idealized 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.

The text of this chapter is in part a reprint of the material as it will
appear in the proceedings of the 15th Workshop on Languages and Compilers for
Parallel Computing (LCPC). The dissertation author was the primary researcher
and author for this publication, which forms the basis for this chapter.
Chapter 4

Using Sparse Tiling in Existing Software Packages

The previous chapter describes experiments involving a sparse-tiled Gauss-Seidel kernel. In this chapter, we describe using sparse tiling within two existing software packages: the Finite Element toolkit (FEtk) [60] and a more general scientific computation library, the Portable, Extensible Toolkit for Scientific Computation (PETSc) [11]. In Section 4.1, we use FEtk to explore the technical issues involved in using sparse-tiled Gauss-Seidel within the context of symmetric multigrid. In Section 4.2, we implement a sparse tiling extension to PETSc to explore the software engineering issues involved in incorporating run-time re-ordering transformations into existing software packages. For both sparse-tiled Gauss-Seidel used within the context of symmetric multigrid and sparse-tiled SOR used within the context of the Krylov solver GMRES, we experiment with different options for amortizing the sparse tiling inspector overhead.
4.1 Sparse Tiling in FEtk

FEtk is a collection of software tools for solving partial differential equations using finite element analysis. In our experiments we use the Manifold Code (MC) finite element code from this collection. Multigrid methods are frequently used in Finite Element Analysis to solve simultaneous systems of linear equations. This is because the asymptotic performance of multigrid is optimal, $O(R)$ where $R$ is the number of unknowns, when used with regular grids [48], and the asymptotic performance has been shown optimal on some model problems for irregular grids [3]. Stationary iterative methods, or smoothers [18], are used at each level of multigrid. Typically, the smoother dominates the execution time of multigrid methods [3], therefore presenting an opportunity for full sparse tiling transformations to improve overall multigrid performance. Since multigrid methods call a smoother on the same matrix multiple times, it is possible to amortize the inspector overhead.

We specifically investigate using sparse-tiled Gauss-Seidel with symmetric multigrid. FEtk provides functionality for generating a hierarchy of meshes and performing finite element analysis to generate the corresponding systems of linear equations. With this functionality, we address two technical issues involved in using sparse-tiled Gauss-Seidel with symmetric multigrid: first, how the interface between finite element analysis and multigrid affects sparse tiling overhead, and second, how to adapt the data-reordering function and sparse tiling schedule for reverse Gauss-Seidel.

4.1.1 Interface between Finite Element Analysis and Multigrid

Finite element analysis (FEA) is a numerical technique for solving partial differential equations in scientific applications. Some example applications include stress analysis, heat transfer, and fluid flow. The physical domain is
modeled with finite elements, such as triangles and tetrahedrons, that form an irregular grid or mesh. FEA generates simultaneous linear equations that describe the relationship between the unknowns at each node in the mesh. Typical unknowns include xy-displacement, temperature, and pressure. These equations are represented as a systems of simultaneous linear equations \( A \vec{u} = \vec{f} \), where \( \vec{u} \) is a vector of unknowns, \( A \) is a sparse matrix of coefficients, and \( \vec{f} \) is a vector of known constants referred to as the right-hand side.

Stationary iterative methods such as Gauss-Seidel, Successive Overrelaxation (SOR), and Jacobi can be used to iteratively solve a system of linear equations. However, although stationary iterative methods are effective at iteratively reducing the high-frequency error in the estimated solution, they are ineffective at reducing the low-frequency error, causing slow convergence rates. Multigrid methods improve the convergence rate by mapping the low-frequency error of a fine-grained mesh onto a coarser-grained mesh. This turns the low-frequency error into high-frequency error in the smaller system of equations.

The pseudocode shown in Figure 4.1 corresponds to the multigrid ‘V-cycle’ as it is implemented in FEtk. Figure 4.2 illustrates how the recursive nature of the multigrid algorithm results in a ‘V’ where each level of the picture corresponds to a particular instantiation of the MULTIGRID algorithm. Here we assume that before MULTIGRID is called, a hierarchy of matrices, \( A_0, ..., A_{n-1} \), has already been generated. MULTIGRID is first called on the largest matrix \( A_{n-1} \) corresponding to the system of equations for the unknowns in the finest mesh. For this discussion, assume the smallest matrix is \( A_0 \) and is small enough for a direct solve. The MULTIGRID algorithm operating on \( A_{\text{level}} \) recursively calls itself on the next smaller sparse matrix \( A_{\text{level-1}} \). This process repeats until the system of linear equations associated with the smallest matrix \( A_0 \) is solved with a direct solver, such as Gaussian Elimination using LU factorization. The results from
MULTIGRID(level, \vec{u}, \vec{f}, MG_{itmax})

1: if \( A_{level} \) is smallest sparse matrix then solve with direct solver
2: \( \vec{r} = A_{level} \vec{u} - \vec{f} \)
3: while (\( iters < MG_{itmax} \)) and (\( \text{norm}(r) > etol \))
4: call GaussSeidelCSR(\( A_{level}, \vec{u}, \vec{f} \))
5: \( \vec{r} = \vec{f} - A\vec{u} \)
6: \( \vec{f}_c = P^T_{level} \vec{r} \)
7: call MULTIGRID(level - 1, \vec{e}_c, \vec{f}_c, 1)
8: \( \vec{e} = P_{level} \vec{e}_c \)
9: \( \vec{u} = \vec{u} - \vec{e} \)
10: call ReverseGaussSeidelCSR(\( A_{level}, \vec{u}, \vec{f} \))
11: \( iters = iters + 1 \)
12: end while

Figure 4.1: Symmetric multigrid algorithm

the direct solve are projected back up the hierarchy of matrices.

There are two types of multigrid methods based on how the hierarchy of sparse matrices is generated. Geometric multigrid methods require that a mesh or grid is generated for each level of the matrix hierarchy. In the context of FEA, discretizing the PDE over each mesh results in a hierarchy of systems of linear equations and their associated sparse matrices. Algebraic multigrid methods typically only require the largest system of linear equations and associated sparse matrix. These methods generate the hierarchy of smaller matrices using various methods [3].

Designing the data structures and associated access methods for performing FEA and using multigrid for the linear solver is complex. In [108], Rüde suggests that the interface between FEA and multigrid should be a hierarchy of meshes. However, this assumes that there is a mesh data structure available for each level of multigrid, which is not the case with algebraic multigrid methods [3]. Such methods generate a sparse matrix at each multigrid level without generating the associated irregular grids or meshes.
Figure 4.2: A symmetric multigrid 'V'-cycle on the left and an associated matrix hierarchy shown on the right.

In our experiments, we assume a hierarchy of sparse matrices is created and then used by the MULTIGRID algorithm. This is the most general situation, since both geometric multigrid methods and algebraic multigrid methods generate a hierarchy of matrices. Besides the sparse matrices that represent the systems of equations at each level of the multigrid hierarchy, there are also the prolongation and restriction matrices that map unknown and right-hand side vectors between levels (see Figure 4.2). The prolongation matrices are created such that $A_{\text{level}-1} = P_{\text{level}}^T A_{\text{level}} P_{\text{level}}$, where $P^T$ is the transpose of $P$.

It is practical to store the hierarchy of sparse matrices after they have been generated and use them for multiple calls to multigrid. One instance where
calls occur to the same sparse matrix hierarchy multiple times is when the right hand side of the system of equations for the finest mesh, $A_{n-1}\vec{u} = \vec{f}$, changes. Multiple calls involving the same matrix also occur when using Newton methods for linearization of non-linear systems [61] and when solving parabolic systems since the system is evolving over time.

In our experiments, we explore the multigrid solve performance when the sparse tiling inspector is included in the initial creation of the matrix hierarchy, and when the sparse tiling inspector is included within the time for the multigrid solve. Multigrid calls a smoother like Gauss-Seidel on every level of the matrix hierarchy except for $A_0$. The overhead of applying sparse tiling transformations to the smoother calls in multigrid is two-fold. The first and largest part of the overhead is the sparse tiling inspector. The sparse tiling inspector overhead includes partitioning the matrix graphs for $A_1, \ldots, A_{n-1}$, growing tiles for the iteration spaces induced by each matrix graph, generating schedule functions for use by Gauss-Seidel, generating a data-reordering functions, and finally remapping each matrix based on its own data-reordering function. The sparse tiling inspector must also remap the prolongation matrices. The rows in $P_{\text{level}}$ are remapped using the same data reordering function used to remap $A_{\text{level}}$. The columns in $P_{\text{level}}$ are remapped using the data reordering function for $A_{\text{level}-1}$. The second part of the overhead is a data remapping upon entry and exit of the \textsc{Multigrid-ST} algorithm, as shown in lines 3 and 17 of Figure 4.3.

Another issue that affects the implementation of multigrid is the sparse matrix format. Considering the benefits of various sparse matrix formats is outside the scope of this dissertation. We use the CSR representation because it is commonly used. Although symmetric representations save storage space, representations such as CSR in which the nonzeros for a particular row are easily accessed are more efficient for smoothers like Gauss-Seidel [108].

\footnote{$A_0$ is not sparse tiled because it will be solved with a direct solver}
Figure 4.3: Symmetric multigrid algorithm for use with sparse-tiled Gauss-Seidel

4.1.2 Using Sparse Tiling in Symmetric Multigrid

Multigrid can be used as a stand-alone solver or as a preconditioner to Krylov methods such as Conjugate Gradient. Conjugate Gradient requires a symmetric preconditioner, therefore, symmetric multigrid is an important multigrid variant. Symmetric multigrid uses Gauss-Seidel with forward ordering on the way down the V-cycle and reverse Gauss-Seidel on the way up (see Figures 4.1 and 4.2). Therefore, we need to use the same data reordering function and preferably the same schedule function for the sparse-tiled versions of both the forward and reverse Gauss-Seidel. The pseudocode for sparse-tiled reverse Gauss-Seidel is shown in Figure 4.4.

One can visualize the typical schedule for reverse Gauss-Seidel as executing the convergence iterations for forward Gauss-Seidel in reverse, and within
ReverseGaussSeidel\((A,\vec{u},\vec{f})\)
for tileID\(_r\) = 0 to \((k - 1)\) do
  for iter\(_r\) = 1 to \(T\) do
    for \(i \in \text{sched}((k - 1) - \text{tileID}_r, (T + 1) - \text{iter}_r)\) in reverse order
      1: \(u'[i] = f'[i]\)
      for \(p = i\text{a}[i + 1] - 1\) downto \(i\text{a}[i]\) do
        if \((\text{ja}[p] \neq i)\) then
          \(u'[i] = u'[i] - a[p] * u'[ja[p]]\)
        else
          \(\text{diag}[i] = a[p]\)
      endfor
      endfor
    endfor
  endfor
endfor

Figure 4.4: Serial execution of sparse-tiled reverse Gauss-Seidel for compressed sparse row (CSR)

each convergence iteration executing the iteration points in reverse order by matrix row (see Figures 4.5 and 4.6). Notice that in Figure 4.5 the last tile (covering iteration points in the bottom right quadrant of the matrix graph instances) is an inverse pyramid. Therefore, if reverse Gauss-Seidel is logically executing the convergence iterations in reverse, this last tile for forward sparse-tiled Gauss-Seidel is the first tile for reverse sparse-tiled Gauss-Seidel. We implement this by indexing the schedule function sched with the reverse tile \(((k - 1) - \text{tileID}_r)\) and the reverse convergence iteration \(((T + 1) - \text{iter}_r)\). By formalizing this intuition we are able to show that the new schedule results in a legal schedule for reverse Gauss-Seidel.

Theorem 4 Let \(G(V,E)\) be the directed matrix graph for a square sparse matrix \(A\). Let \(\{0,...,(k - 1)\}\) be the set of tiles, \(\{1,...,T\}\) be the set of conver-
Figure 4.5: Full sparse-tiled Gauss-Seidel iteration space with 3 convergence iterations

gence iterations, and \( \{0, \ldots, (R - 1)\} \) be the set of nodes in \( V \). Let \( \theta_r(\text{iter}_r, v) = (k-1)−\theta(T+1−\text{iter}_r, v) \), and let \( \sigma_r(v) = R−1−\sigma(v) \), where \( \theta \) and \( \sigma \) are the tiling and data reordering functions generated by the \text{FullSparseNaive}_GCSR algorithm.

The reverse tiling function \( \theta_r \) satisfies the constraints specified in Theorem 1. Since \( \theta_r \) and \( \sigma_r \) satisfy these constraints, the unknown vector generated by sparse-tiled reverse Gauss-Seidel is bit-wise equivalent to typical reverse Gauss-Seidel assuming the unknowns have the same ordering.

Proof:

In the proof for Theorem 2, the constraints from Theorem 1 are rewritten and shown satisfied by the tiling function \( \theta \) and data-reordering function \( \sigma \)
Figure 4.6: Full sparse-tiled reverse Gauss-Seidel iteration space. The tiling for iteration $\text{iter}_r = 1$ is the same as that for $\text{iter} = (T + 1) - \text{iter}_r = 3$ in Figure 4.5.

generated by the FULLSPARSE_NAIVE_GSCSR algorithm. For the reverse tiling function $\theta_r$ and the reverse data-reordering function $\sigma_r$, the same constraints must be satisfied to show that the reverse Gauss-Seidel is legal. Here we rewrite the constraints in Theorem 2 for $\theta_r$ and $\sigma_r$.

1. $\forall \text{iter}_r, 1, \text{iter}_r, v$ such that $(0 \leq \sigma_r(v) < R)$, if $(1 \leq \text{iter}_r < \text{iter}_r \leq T)$, then $\theta_r(\text{iter}_r, v) \leq \theta_r(\text{iter}_r, v)$

2. $\forall \text{iter}_r, v, w$ such that $(1 \leq \text{iter}_r \leq T)$, if $(0 \leq \sigma_r(v) < \sigma_r(w) < R)$ and $\langle v, w \rangle \in E$, then $\theta_r(\text{iter}_r, v) \leq \theta_r(\text{iter}_r, w)$

3. $\forall \text{iter}_r, v, w$ such that $(1 \leq \text{iter}_r \leq T)$, if $(0 \leq \sigma_r(v) < \sigma_r(w) < R)$ and $\langle w, v \rangle \in E$, then $\theta_r(\text{iter}_r, v) \leq \theta_r(\text{iter}_r, w)$
4. \( \forall \ iter_{r1}, \ iter_{r2}, v, w \) such that \( 1 \leq \ iter_{r1} < \ iter_{r2} \leq T \), if \( 0 \leq \sigma_r(v) \neq \sigma_r(w) < R \) and \( <v, w> \in E \), then \( \theta_r(\ iter_{r1}, v) \leq \theta_r(\ iter_{r2}, w) \)

5. \( \forall \ iter_{r1}, \ iter_{r2}, v, w \) such that \( 1 \leq \ iter_{r1} < \ iter_{r2} \leq T \), if \( 0 \leq \sigma_r(v) \neq \sigma_r(w) < R \) and \( <w, v> \in E \), then \( \theta_r(\ iter_{r1}, v) \leq \theta_r(\ iter_{r2}, w) \)

For Constraint 1, we substitute \( \theta_r \) and \( \sigma_r \) with the relationships stated in this theorem to derive the following constraint:

\[
\forall \ iter_{r1}, \ iter_{r2}, v \text{ such that } (1 \leq \ iter_{r1} < \ iter_{r2} \leq T), \quad \\
\text{if } (0 \leq R - 1 - \sigma(v) < R), \\
\text{then } (k - 1) - \theta(T + 1 - \ iter_{r1}, v) \leq (k - 1) - \theta(T + 1 - \ iter_{r2}, v).
\]

Through simplification and by letting \( \ iter_a = T + 1 - \ iter_{r1} \) and \( \ iter_b = T + 1 - \ iter_{r2} \), the constraint can be rewritten as the following:

\[
\forall \ iter_a, \ iter_b, v \text{ such that } (1 \leq \ iter_b < \ iter_a \leq T), \quad \\
\text{if } (0 \leq \sigma(v) < R), \\
\text{then } \theta(\ iter_b, v) \leq \theta(\ iter_a, v),
\]

which is the Constraint 1 that was shown true for the \( \theta \) and \( \sigma \) in Theorem 2.

Constraints 2 through 5 can be shown with the same substitutions and correspondence to constraints on \( \theta \) and \( \sigma \).

4.1.3 Using FEtk to Generate the Matrix Hierarchy

We use the MC (Manifold Code) portion of the FEtk software package to generate the matrix hierarchy. We also change MC’s symmetric multigrid so that it calls a sparse-tiled Gauss-Seidel and reverse Gauss-Seidel. At the time of these experiments, MC only has a geometric multigrid implementation. To generate the hierarchy of matrices specified in Table 4.1, we have MC uniformly
<table>
<thead>
<tr>
<th>Domain</th>
<th>Matrix</th>
<th>rows</th>
<th>nonzeros</th>
<th>nonzeros/rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D Bar</td>
<td>$A_1$</td>
<td>19,683</td>
<td>778,473</td>
<td>39.55</td>
</tr>
<tr>
<td></td>
<td>$A_0$</td>
<td>3,075</td>
<td>108,873</td>
<td>35.41</td>
</tr>
<tr>
<td>Pipe</td>
<td>$A_2$</td>
<td>381,120</td>
<td>15,300,288</td>
<td>40.15</td>
</tr>
<tr>
<td></td>
<td>$A_1$</td>
<td>57,312</td>
<td>2,114,784</td>
<td>36.90</td>
</tr>
<tr>
<td></td>
<td>$A_0$</td>
<td>9,600</td>
<td>315,072</td>
<td>32.82</td>
</tr>
<tr>
<td>2D Bridge</td>
<td>$A_3$</td>
<td>367,706</td>
<td>5,073,588</td>
<td>13.78</td>
</tr>
<tr>
<td></td>
<td>$A_2$</td>
<td>94,234</td>
<td>1,282,356</td>
<td>13.61</td>
</tr>
<tr>
<td></td>
<td>$A_1$</td>
<td>24,698</td>
<td>327,540</td>
<td>13.26</td>
</tr>
<tr>
<td></td>
<td>$A_0$</td>
<td>6,730</td>
<td>85,332</td>
<td>12.68</td>
</tr>
</tbody>
</table>

Table 4.1: Descriptions of input matrices used in multigrid experiments

refine a mesh to generate fine-grained meshes from course-grain meshes, which are provided as examples in FEtk: 3D Bar, 3D Pipe, and 2D Bridge. The 2D meshes consist of triangles, and the 3D meshes consist of tetrahedrons. For the triangles, we use uniform quadra-section, and for the tetrahedrons we use uniform octa-section to refine the meshes. For each mesh granularity, we solve a linear elasticity boundary problem, as described in Section 2.5, to generate a matrix for each mesh in the hierarchy. Each time refinement is performed on a mesh, MC generates the prolongation matrices needed to restrict and prolong unknown vectors between levels in the hierarchy.

4.1.4 Experimental Results with Symmetric Multigrid

The experiments are run on a 250MHz MIPS R10000 processor, which is summarized in Table 4.2. The ‘cc’ compilation command invokes the MIPS-pro compiler. The symmetric multigrid implementation iterates V-cycles until a relative residual of 1e-6 is achieved.

First, we report the scenario where the sparse tiling inspector overhead is included in the initial construction of the sparse matrix hierarchy and not included in the multigrid solve time. Figures 4.7 and 4.8 show the normalized
<table>
<thead>
<tr>
<th>Chip</th>
<th>Description</th>
<th>MHz</th>
<th>L1 cache</th>
<th>L2 cache</th>
<th>compilation command</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10000</td>
<td>Node of SGI Origin 2000</td>
<td>250MHz</td>
<td>32KB</td>
<td>4MB</td>
<td>cc -O3 -DNDEBUG</td>
</tr>
</tbody>
</table>

Table 4.2: Descriptions of computer architecture used in symmetric multigrid experiments

execution times of symmetric multigrid using sparse-tiled Gauss-Seidel compared with regular symmetric multigrid. In Figure 4.7, each time Gauss-Seidel or reverse Gauss-Seidel is called, two convergence iterations occur. In Figure 4.8, four convergence iterations occur each time either Gauss-Seidel routine is called. We experiment with both the cache blocking and full sparse tiling transformations. For both transformations, we show the results when the tiles fit into 1/2 the L2 cache and 1/8 of the L2 cache.

Both sparse tiling transformations consistently result in a faster execution for symmetric multigrid. There are many observations we can make with respect to Figures 4.7 and 4.8. First, increasing the number of convergence iterations in the Gauss-Seidel smoother definitely increases the benefit of sparse tiling transformations for these matrices. This is understandable because more convergence iterations have more potential for inter-iteration locality. Second, both sparse tiling transformations perform slightly better at 2 convergence iterations when they target seed partition size to fit into 1/8 of L2 cache. The performance of the cache-blocked code with matrices resulting from 3D meshes deteriorates when tiling for 1/8 of L2 cache with 4 convergence iterations in the Gauss-Seidel smoother. This is probably due to having more tiles coupled with more convergence iterations in the cache blocking transformation. More tiles result because smaller seed partitions are needed to fit within 1/8 of L2 cache versus 1/2 of L2 cache. More tiles results in a larger final tile, and the final tile in cache-blocked Gauss-Seidel can exhibit poor data locality. Previously, in [35] speedups between
Figure 4.7: Normalized execution time of sparse-tiled symmetric multigrid compared with regular symmetric multigrid. The forward and reverse Gauss-Seidel smoothers do 2 convergence iterations at each multigrid level.

1.75 and 2.07 were reported for a 3 level multigrid V-cycle where the largest sparse matrix had 124,414 rows. In our experiments, both sparse tiling transformations achieved speedups between 1.09 and 1.43. The discrepancy is probably due to the fact that we are not calculating the residual with the Gauss-Seidel tiles as [35] does, and we are using a different multigrid implementation.

In Figure 4.9, we show the normalized TLB misses, L2 cache misses, and L1 cache misses obtained using PAPI [82], a software package that provides an interface to hardware counters. When tuned for the L2 cache size, sparse tiling transformations most consistently reduce the number of L2 cache misses in symmetric multigrid with little effect on L1 cache misses. The effect on the TLB misses seems to vary with each problem and sparse tiling transformation. One result to note is that for the larger 3D Pipe problem, the full sparse tiling
Figure 4.8: Comparing the normalized execution time of sparse-tiled symmetric multigrid with regular symmetric multigrid. The forward and reverse Gauss-Seidel smoothers do 4 convergence iterations at each multigrid level.

Figure 4.9: Comparing the normalized TLB, L2, and L1 cache misses of sparse-tiled symmetric multigrid with regular symmetric multigrid. The forward and reverse Gauss-Seidel smoothers do 4 convergence iterations at each multigrid level. Both transformations consistently reduce the number of L2 cache misses.
Figure 4.10: Normalized execution time of sparse-tiled symmetric multigrid, with sparse tiling overhead included, compared with regular symmetric multigrid. Both the forward and reverse Gauss-Seidel smoothers do 2 convergence iterations at each multigrid level.

...transformation results in significant increase in the number of TLB misses. Other research on regular grid cache optimizations has also observed the increased importance of TLB misses in 3D discretizations [78, 107].

Finally, we look at the scenario in which the sparse tiling inspector is called upon entry to the multigrid function. This results in all the overhead being included in the solution time. Figures 4.10 and 4.11 show the normalized execution time when the overhead of sparse tiling inspector for all levels of the matrix hierarchy is included. Figure 4.10 shows results for the case that 2 convergence iterations are performed by Gauss-Seidel, and Figure 4.11 shows the results for 4 Gauss-Seidel convergence iterations. Some cases result in a slow-down, however this slow-down is minimal and can be amortized over 2 or 3 more calls to
Figure 4.11: Normalized execution time of sparse-tiled symmetric multigrid, with the sparse-tiling inspector overhead included, compared with regular symmetric multigrid. The forward and reverse Gauss-Seidel smoothers do 4 convergence iterations at each multigrid level.

symmetric multigrid on the same sparse matrix hierarchy.

4.2 Sparse Tiling in PETSc

We also implement full sparse tiling as an extension to the PETSc package [11, 12, 13]. PETSc is a parallel numerical software package developed for scalable/parallel computing systems. Also, PETSc provides a mechanism for extending the matrix data structure and specifying special functions such as sparse-tiled SOR (Gauss-Seidel is a special case of SOR). Our extension called STPetsc is available on the web [118].

The PETSc library handles many implementation issues related to per-
formance such as parallelization, sparse matrix formats, and usage levels. We explore how the run-time reordering transformation, full sparse tiling, might be incorporated while maintaining the established software architecture. The initial implementation of our STPetsc extension includes two interface levels, a beginner interface where the user specifies only that sparse tiling should be applied, and an intermediate interface where decisions involving when the sparse tiling inspector executes are exposed to the user. Experiments show that it is possible to amortize the overhead of the STPetsc inspector within a GMRES solve preconditioned with sparse-tiled SOR when using the intermediate interface.

4.2.1 PETSc Overview

PETSc is maintained and developed by researchers at Argonne National Labs. It is used in many applications areas such as Optimization, Biology, and Finite Element Analysis. One of the main objectives for the PETSc package is to hide the message passing details needed to develop scalable scientific applications. PETSc is implemented in the C programming language and can be used by scientific applications written in C, C++, or Fortran. The main components in the PETSc library are matrices, vectors, linear solvers, and non-linear solvers.

PETSc handles many of what is referred to in the software engineering literature as cross-cutting aspects [75]. In object-oriented programming the problem domain is modeled in software by decomposing concepts into objects with associated methods. Cross-cutting aspects are concepts that involve code in many or all of the objects. A common example of this is logging code. One type of logging information might be function entries and exits. Besides parallelism, the PETSc software architecture is organized to handle cross-cutting aspects such as logging, profiling, visualization, various sparse matrix formats, and a variety of numerical algorithms. It is possible to select and parameterize many numerical
algorithms such as Krylov solvers and various preconditioners. All solvers that fall within a particular category, such as Krylov solvers, are invoked through one interface and can be selected dynamically through the use of function pointers, an implementation mechanism for polymorphism.

One of the cross-cutting aspects within the PETSc package is the usage level. PETSc is an example of an open implementation [74], in that the beginner user interface exposes many tasks such as solving a system of linear equations in a black box fashion, the intermediate user interface allows the user to specify which algorithms and parameters should be used for such tasks, and the advanced user interface allows the user to extend PETSc by implementing his or her own algorithms for such tasks.

The drawback of extending PETSc with a new aspect like the sparse tiling transformation for SOR is that each new aspect has a multiplicative effect on the number of algorithm implementations. This occurs because in general algorithms that traverse matrices, such as matrix-vector multiply and stationary iterative methods, must be implemented separately for each sparse matrix format. We decided to initially restrict our implementation to sparse tiling for the SOR preconditioner operating on matrices in the compressed sparse row matrix format (referred to as sequential AIJ in PETSc). SOR is equivalent to Gauss-Seidel when the extrapolation factor equals one [14].

In PETSc, systems of linear equations are solved either with direct methods or iteratively with preconditioned Krylov subspace methods. Conjugate Gradient and Generalized Minimal Residual (GMRES) are examples of Krylov methods [110, 32]. Krylov methods invoke a preconditioner at each iteration to improve the condition of the matrix and therefore accelerate convergence. In our experiments, we use PETSc’s implementation of GMRES, because Conjugate Gradient requires a symmetric preconditioner. Our initial STPetsc extension only
includes sparse-tiled SOR, which is not a symmetric preconditioner.

4.2.2 Possibilities for Sparse Tiling SOR

Sparse tiling transformations, such as cache blocking [35] and full sparse tiling, create an iteration reordering for stationary iterative solvers and a data reordering for the sparse matrix A, unknown vector \( \vec{u} \), and right-hand side vector \( \vec{f} \). The input to the sparse tiling inspector is a matrix, the number of tiles, and a number of convergence iterations for which tiles should be grown. The output is a data-reordering function and a schedule for use within the sparse tiling executor. The initial STPetsc implementation includes a sparse tiling inspector and executor for SOR operating on the compressed sparse row (CSR) matrix format (referred to as sequential AIJ within PETSc package). The tile growth algorithm used is full sparse tiling.

The scenarios for how sparse tiling can be used with PETSc’s SOR fall along two main axes: whether or not the sparse matrix and/or vectors are remapped based on the data reordering function generated by the sparse tiling inspector, and when they are remapped with respect to the rest of the program. First, we enumerate how the data structure status in terms of remapping will affect a sparse-tiled execution of SOR. Then, we will describe the various points at which remapping is possible within the context of a Krylov solver preconditioned with SOR, and how executing the inspector at those points will affect run-time overhead.

**Effect of Data Remapping**

Whether or not the unknown and right-hand side vectors are remapped according to the data-reordering function generated by the sparse tiling inspector affects the overhead of the inspector, the number of indirect memory references in the sparse-tiled SOR executor, and the executor data locality. If the unknown
and right-hand side vectors are not remapped there are more indirect memory references and poorer spatial locality. Therefore, even though remapping the vectors does cause more overhead in the inspector, in our STPetsc extension we remap the vectors.

Whether or not the matrix is remapped also affects the overhead cost of the sparse tiling inspector, the number of indirect memory references in executor, and the executor data locality as summarized in Table 4.3. In addition, the status of the matrix affects the roundoff error and the necessity of changes to the PETSc SOR implementation due to existing performance optimizations. For both the possibilities of remapping the matrix we assume that the matrix is in the compressed sparse row (CSR) format.

If the sparse matrix $A$ is NOT remapped, the inspector requires much less time because remapping sparse matrices is expensive. This option also saves space because it is not necessary to maintain two copies of the matrix. If the sparse matrix is not remapped, the number of indirect memory references in the computation increases when the code accesses the rows and uses the column identification for the nonzeros to access into the remapped unknown and right-hand side vectors. The spatial locality that occurs by using rows in the sparse matrix sequentially may be suboptimal, since the new ordering will skip around in the old ordering of rows. However, there will be spatial locality while visiting the nonzeros within a row.
Not remapping the matrix also complicates the roundoff error and requires changes to the PETSc SOR implementation. The roundoff error is different than in either the non-remapped original or the remapped sparse tiling because the nonzeros within each row are in the old order, and the rows are visited in the new order. The current implementation of SOR in PETSc uses the order of nonzeros in each row of the sparse matrix $A$ to avoid zeroing out the unknown vector when an initial guess of zero should be used for the unknown vector. Since the nonzeros in the original matrix are not in the new order assumed by the sparse tiling schedule, the sparse-tiled SOR executor must zero out the unknown vector in the case of an initial guess of zero. The computation must also visit all the nonzeros in the row and not just those before the diagonal as is currently implemented in PETSc’s SOR as a performance optimization. Both of these requirements cause some extra work for the executor.

The other option is to create a new sparse matrix $A'$ that is a remapped version of the original sparse matrix $A$. This option is the most expensive in terms of the inspector overhead. Remapping an entire sparse matrix requires a lot of storage and time. The number of indirect memory references is the same as what is in the current SOR implementation. In addition there is some indirect indexing in the loop bounds over the unknowns due to the sparse tiling schedule data structure. The locality of references in the sparse matrix is almost as good as the original because rows are stored consecutively by the new ordering. The locality in the accesses to the remapped unknown and right-hand side vectors are much improved.

Using the remapped matrix only requires the insertion of the extra tiling loop and references to the sparse tiling schedule data structure into PETSc’s SOR implementation, which is necessary for either matrix remapping option. The computation results in a different rounding error than the original ordering, but
### Table 4.4: Summary of the effect of where the sparse tiling inspector is located

<table>
<thead>
<tr>
<th>Where ST inspector is invoked</th>
<th>user interface</th>
<th>overhead</th>
<th>locality throughout solve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before entire solve</td>
<td>more complex</td>
<td>less</td>
<td>better</td>
</tr>
<tr>
<td>Upon first SOR call</td>
<td>easy</td>
<td>more</td>
<td>SOR good, rest of solve the same</td>
</tr>
</tbody>
</table>

the same rounding error as the original SOR if the original SOR uses the same data reordering as sparse-tiled SOR.

Based on the different effects of the two possibilities, we chose to remap the entire matrix as well as the vectors.

**Effect of Inspector Location**

Within the context of a Krylov solver preconditioned with SOR there are two main points when the sparse tiling inspector (including remapping of the matrix and vectors) may be executed: before the entire solve (before the `SLESSolve()` call) or the first time that SOR is called. We initially assume that the matrix’s sparsity structure and values stay the same throughout the entire solve. The vectors passed to the SOR function will vary, therefore if the reordered matrix is only used within SOR, it is necessary to reorder the unknown and right-hand side vector upon entry to the SOR function, and reorder the unknown vector back on the way out.

When the sparse tiling inspector is called affects the user interface to sparse tiling, the overhead caused by data remappings, and the locality experienced within the solve. These issues are summarized in Table 4.4.

Invoking the sparse tiling inspector before the entire solve requires the user to take responsibility for calling the sparse tiling inspector. In terms of overhead, if the original order on the data structures must be maintained, then
it is necessary to store two matrices unless it is more efficient to reconstruct the originally ordered matrix. If the original matrix ordering is no longer needed, then the user can allow the rest of the program to use the newly ordered matrix and delete the original matrix. Since the sparse matrix and vectors are reordered before the entire solve, all of the algorithms involved in the solve benefit from better locality when accessing the vectors based on the nonzero structure in the matrix.

Invoking the sparse tiling inspector the first time SOR is called allows the invocation of the sparse tiling inspector to be hidden from the user, because the rewritten SOR can incorporate a call to the inspector. The user need only indicate that sparse-tiled SOR is the desired preconditioner. The sparse tiling schedule and remapped sparse matrix can be memoized the first time SOR is called. In this scenario both matrices must be maintained because SOR is using the remapped matrix, and the rest of the solve is using the original matrix. There is also more overhead in this scenario because both vectors must be remapped upon entry to SOR, and upon exit the unknown vector must be remapped back to the original ordering. Only the SOR function benefits from the improved spatial locality due to the matrix and vector remappings.

We allow for both locations for inspector invocation by implementing the beginner and intermediate user interfaces. In our experiments, we look at how usage of the two different interfaces affects performance.

4.2.3 STPetsc Two-Level Interface

We designed the sparse tiling extension to PETSc with the following goals in mind: the extension should follow the PETSc software architecture, be easy to use, and provide intermediate users with sufficient control.

The current concepts in PETSc that the sparse tiling inspector and
executor most closely resemble are Partitionings, and special implementations of certain matrix functions such as the ESSL implementation of LU factorization and solve. Partitionings are associated with and manipulate a specific matrix. The sparse tiling inspector performs a partitioning of a matrix and then grows tiles for a constant number of convergence iterations. The sparse tiling executor for SOR is a special implementation of the MatRelax function.

The beginner interface to STPetSc only requires the user to call the function MatUseST_SeqAIJ(). In the beginner interface, the first time SOR is called the sparse tiling inspector is invoked and the sparse tiling schedule, data reordering function, and newly remapped sparse matrix is memoized. The SLESSolve() function, in PETSCc’s beginner interface, performs a solve for a system of linear equations, $A\vec{u} = \vec{f}$. Options concerning the type of Krylov solver, type of preconditioner, maximum number of allowed iterations, convergence tolerances, etc. can be set with command-line parameters to the program using PETSc. The command-line parameters ‘-pc_type sor’ causes the sparse-tiled SOR to be used as the preconditioner as long as the MatUseST_SeqAIJ() function is called within the program as well.

The intermediate interface allows the user to specify which matrix should be associated with a sparse tiling, the specification of parameters for the sparse tiling, the sparse tiling to be written and loaded from a file, and the sparse tiling inspector to be invoked before the first call to SOR (eg. before a call to SLESSolve()).

The intermediate user has access to the following functions:

- `MatSparseTilingCreate()` - creates a sparse tiling context
- `MatSparseTilingSetMatrix()` - associates a matrix with the sparse tiling
- `MatSparseTilingSetFromOptions()` - captures sparse tiling parameters from the command line
• **MatSparseTilingApply()** - generates a schedule and data reordering function and stores them with the associated matrix

• **MatSparseTilingRemap()** - This function does an in place remapping on the unknown vector and the right-hand side vector. A new matrix is created that is the remapped version of the matrix associated with the MatSparseTiling. We set a flag on the new matrix to indicate that it is the remapped matrix, therefore when SOR is called a new remapped matrix is not created.

Other functions that will be implemented in the future include:

• **MatSparseTilingView()** - write the sparse tiling schedule, data reordering function, and remapped matrix to a file

• **MatSparseTilingLoad()** - read the sparse tiling schedule, data reordering function, and remapped matrix from a file

### 4.2.4 Experimental Results Using STPetsc

Often systems of linear equations are solved numerically with preconditioned Krylov solvers. The sparse tiling inspector/executor provided by STPetsc can be used with any Krylov solver that is preconditioned with SOR. This section describes experimental results using the Krylov solver GMRES preconditioned with SOR.

The goal of these experiments is to determine the overhead of the sparse tiling inspector, the effect of sparse tiling on the execution time for each GMRES iteration, and the effect of the data reordering due to graph partitioning or sparse tiling on the residual, \( \| A\tilde{u} - \tilde{f} \|. \) We also examine the performance difference between the beginner and intermediate user interface.

We record two execution times, the wallclock time for the solve setup and that for the solve itself (preconditioned GMRES). The setup includes a
Table 4.5: Descriptions of computer architectures used in STPetsc experiments

<table>
<thead>
<tr>
<th>Chip</th>
<th>MHz</th>
<th>L1 cache</th>
<th>L2 cache</th>
<th>compilation command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4 (2GHz)</td>
<td>2000</td>
<td>8KB</td>
<td>512KB</td>
<td>gcc -O3 -DNDEBUG</td>
</tr>
</tbody>
</table>

Table 4.6: Descriptions of input matrices used in STPetsc experiments

<table>
<thead>
<tr>
<th>Matrix</th>
<th>rows</th>
<th>nonzeros</th>
<th>avg nonzeros per row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wing22K</td>
<td>22,266</td>
<td>923,925</td>
<td>41.49</td>
</tr>
<tr>
<td>2D Bar</td>
<td>74,926</td>
<td>1,037,676</td>
<td>13.85</td>
</tr>
<tr>
<td>Sphere150K</td>
<td>154,938</td>
<td>11,508,390</td>
<td>74.28</td>
</tr>
</tbody>
</table>

Table 4.7: The number of GMRES iterations needed for GMRES preconditioned with the original PETSc SOR implementation to converge with the specified preconditioned relative tolerance (rtol).

<table>
<thead>
<tr>
<th>Matrix</th>
<th>rtol</th>
<th>SOR convergence iterations</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wing22K</td>
<td>1e-1</td>
<td></td>
<td>118</td>
<td>78</td>
<td>36</td>
<td>28</td>
</tr>
<tr>
<td>2D Bar</td>
<td>1e-1</td>
<td></td>
<td>59</td>
<td>177</td>
<td>376</td>
<td>537</td>
</tr>
<tr>
<td>Sphere150K</td>
<td>1e-4</td>
<td></td>
<td>4107</td>
<td>1622</td>
<td>965</td>
<td>660</td>
</tr>
</tbody>
</table>
Table 4.7, when 1, 2, 3, or 4 convergence iterations of the normal PETSc SOR is executed as the preconditioner. We use that same number of GMRES iterations when the preconditioner is switched to SOR with data reordering only or sparse-tiled SOR, which executes a new tile-by-tile schedule as well as operating on the remapped data.

GMRES executes on a Pentium 4 running at 2GHz with 512KB of L2 cache (see Table 4.5). The input matrices used are listed in Table 4.6. The origin
of these matrices is described in Section 2.5, but in general all three matrices result from Finite Element Analysis. The seed partitioner is an implementation of the Gpart partitioner with parameters selected as described in [50]. The maximum seed partition size is made to fit into 1/2 of the L2 cache, based on the experimental results in Figure 2.18 that indicate there is not much difference between target seed partition sizes for the Pentium 4 (2GHz). In addition, we pre-load and solve a smaller matrix in order to improve usage of the instruction cache [13].

Figure 4.12 shows the total execution time in seconds for the solve setup and GMRES solve when the original PETSc SOR is used (1st bar), when the data is remapped based on a Gpart data reordering (2nd and 4th bars), and when full sparse tiling is used to reorder the iteration and the data (3rd and 5th striped bars). The results in this figure are for the Wing system of linear equations.
Figure 4.14: Residual norm after a set number of GMRES iterations (see Table 4.7) for the Wing matrix.

The 2nd and 3rd bars represent the beginner user mode where the sparse tiling inspector is executed the first time SOR is called and therefore not accounted for separately in the setup time. Also, when using the beginner interface, the unknown and right-hand side are remapped each time SOR is called. The 4th and 5th bars represent the intermediate user interface where the sparse tiling inspector including data remapping occurs before the entire `SLESSolve()` call and is included separately in the setup time.

The setup (or overhead) time required by the sparse tiling or data re-ordering inspector when the intermediate interface is used is shown as the dark portion at the bottom of the 4th and 5th total time bars in Figure 4.12. For this matrix, the overhead of data remapping and/or full sparse tiling is not amortized within the number of convergence iterations performed by GMRES. The
Figure 4.15: Execution time in seconds for the setup and solve of GMRES pre-conditioned with PETSc SOR and STPetsc SOR on the 2D Bar matrix

The main reason the overhead is not amortized is that the benefit of data remapping and/or full sparse tiling is minimal in the serial SOR executor, as can be seen in the execution time per GMRES iteration in Figure 4.13. From the serial processor performance results in Section 2.6, it is clear that full sparse tiling for Gauss Seidel (a special case of SOR) does not result in a large serial performance benefit. The real performance benefit occurs when the full sparse tiling is used for parallelism, which has yet to be implemented in the STPetsc extension. The data reordering and full sparse tiling transformations do not greatly affect the resulting residual norm for the Wing matrix.

Figure 4.15 shows that the 2D bar matrix does result in an overall performance benefit when more than one SOR iteration is used. This is true even when the overhead of the data reordering and/or full sparse tiling inspector
Figure 4.16: Execution time per GMRES iteration on the 2D Bar Matrix

is included. This performance improvement is shown on a per GMRES iteration basis in Figure 4.16. The time per iteration is calculated as the total solve time divided by the number of GMRES iterations. Since the solve setup time is not included, the 4th and 5th bars, which show results when the intermediate usage mode is being used, experience better performance within the solve. These last two bars indicate what the improvement per GMRES iteration would be if there were infinitely many GMRES iterations to amortize all of the full sparse tiling inspector overhead. As with the Wing matrix, the data reordering and/or full sparse tiling does not affect the final residual norm significantly (see Figure 4.17).

The Sphere matrix results are displayed in Figures 4.18, 4.19, and 4.20. In terms of the total time for the GMRES solve and the performance per GMRES iteration, the sphere matrix has results similar to the Wing matrix, except the setup overhead is a smaller percentage of the overall execution time due to many
more GMRES iterations. The most significant difference between the Sphere matrix and the other two is the effect the data reordering and/or full sparse tiling has on the final residual. The y-axis in Figure 4.20 is shown in logarithmic scale. The reduction in the residual is up to three orders of magnitude better than the original. This is possibly due to the fact that the nodes in the matrix graph are typically ordered highest edge degree first within each seed partition generated with Gpart. This obviously has different effects on different matrices and suggests further investigation as a method for improving the overall convergence rate.

4.3 Related Work

Automating the use of sparse tiling transformations is an important next step for increasing their usefulness. Sparse tiling transformations for Gauss-
Seidel use domain-specific information, specifically that the unknown vector can undergo an *a priori* reordering. Currently a number of research projects are exploring domain-specific compilation. ROSE [101] is a system which generates domain-specific preprocessors. Their framework supports translating a general abstract syntax tree (AST) to a higher-level domain-specific AST, on which domain-specific transformations can then be performed. Interface compilation [37] and telescoping languages [24] look at ways of optimizing uses of library interfaces.

Being able to attach domain-specific semantics to the library interface would allow for the construction of a preprocessor that recognizes when an unknown vector being passed to a Gauss-Seidel function may be *a priori* reordered. The Broadway compiler [16] allows the library expert to specify annotations for

Figure 4.18: Execution time in seconds for the setup and solve of GMRES preconditioned with PETSc SOR and STPetsc SOR on the Sphere matrix
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.

In a similar vein, Griswold *et al.* [46] suggest that the knowledgeable library writer should have access to how programs that use the library are compiled. MAGIK [36] and Open C++ [25] are two systems that provide such functionality. The MAGIK system allows a programmer to take advantage of the semantics of the interface being implemented to do optimizations and other things like error checking on a compiler intermediate representation (IR). The changes made to the IR are then optimized normally by the lcc compiler [39]. Open C++ [25] is a metaobject protocol package for C++. It focuses on lower-level transformations than MAGIK. Specifically, Open C++ allows the programmer to control compilation for class definition, member accesses, virtual function invocation, and
Figure 4.20: Residual norm after a set number of GMRES iterations (see Table 4.7) for the Sphere matrix

Object creation.

Incorporating the sparse tiling transformation into FEtk and PETSc involves handling algorithmic, sparse matrix format, and performance optimization aspects all within the same code. Aspect Oriented Programming [75] recognizes that there are some aspects to software design, such as the above three, that cross-cut entire programs. It would be better to express these aspects in an orthogonal way to the functionality of the program itself. Only generating the needed combination of aspects along with separate interfaces for different aspects reduces code bloat and maintenance costs. In general this is done by designing an aspect language and an aspect weaver that takes the functional description of the program and weaves it with other aspects like sparse matrix formats and performance transformations. Some work has taken an aspect-oriented approach
to combining algorithms and sparse matrix formats [99, 67, 84], and algorithms and performance transformations [89], but not the combination of all three.

Active libraries [126], such as ATLAS [128], PhiPac [17], Blitz++ [125], and FFTW [40], combine algorithms with performance optimizations by controlling how functions in the library are compiled. Another approach for handling performance and algorithms in an orthogonal way is to write algorithms within a specific programming model, such as divide and conquer [43]. The compilation system can then focus on performance strategies for the particular programming model.

4.4 Summary

In this chapter, we experimented with sparse-tiled Gauss-Seidel and sparse-tiled SOR in the context of symmetric multigrid and GMRES. In both situations there are example problems where sparse tiling transformations result in an overall improvement, and examples where it is not possible to amortize the overhead. We address the technical issues involved in the interface between Finite Element Analysis and multigrid, and the need to use the same data ordering for reverse Gauss-Seidel in symmetric multigrid.

Using transformations in software packages such as FEtk and PETSc does require duplicating functionality to create sparse-tiled versions of algorithms such as Gauss-Seidel and SOR. Since maintaining partial duplicates in software packages is difficult and time-consuming, we conclude that sparse tiling transformations along with other performance aspects like sparse matrix formats, should be maintained separately, and necessary combinations generated when needed. This will require compilation methods for automating the application of sparse tiling transformations.
Chapter 5

Run-time Reordering
Transformation Framework

This chapter presents a compile-time framework that allows the explicit composition of run-time data and iteration-reordering transformations. Our framework builds on the iteration-reordering framework of Kelly and Pugh to represent the effects of a given composition. To motivate our extension, we show that new compositions of run-time reordering transformations can result in better performance on three benchmarks.

We show how to express a number of run-time data and iteration-reordering transformations that focus on improving data locality. We also describe the space of possible run-time reordering transformations and how existing transformations fit within it. Since sparse tiling transformations are included in our framework, they become more generally applicable, both to a larger class of applications, and in their composition with other reordering transformations. Finally, within the presented framework data need be remapped only once at runtime for a given composition thus exhibiting one example of overhead reductions the framework can express.
Section 5.1 motivates the composition of run-time reordering transformations by describing experimental results where full sparse tiling composed with other transformations can result in significant performance improvements. Section 5.2 reviews the terminology for the Kelly and Pugh framework using an example irregular kernel (used throughout this chapter). In Section 5.3, we describe how to formally express run-time reordering transformations and the space of possibilities for such transformations. Section 5.4 formally describes how to compose run-time reordering transformations along with an example composed inspector. Section 5.5 shows examples of using the framework to reduce the overhead of run-time reordering transformations. Finally, Sections 5.6 and 5.7 describe related work and summarize this chapter.

5.1 Motivation for Composing Run-time Reordering Transformations

Our experiments compare the performance resulting from various run-time transformation compositions on the \texttt{irreg}, \texttt{nbf}, and \texttt{moldyn} benchmarks. The goal is to motivate a framework that allows compile-time composition of run-time reordering transformations by showing that benchmarks with hand-coded composed transformations result in improved performance. The \texttt{irreg} benchmark exhibits the types of computations found in partial differential equation solvers, the \texttt{nbf} benchmark is abstracted from the GROMOS molecular dynamics code, and the \texttt{moldyn} benchmark is abstracted from the molecular dynamics application CHARMM [50].

Figure 5.1 shows a simplified version of the \texttt{moldyn} benchmark. There is an outer time-stepping loop that makes amortization of run-time reordering overhead possible. Statement \texttt{S1} calculates the new position of a molecule in the \texttt{x} coordinate using the old position, velocity, and acceleration. The \texttt{j} loop
do s = 1 to num_steps
   do i=1 to num_nodes
      S1  x[i] = x[i] + vx[i] + fx[i]
   enddo
   do j=1 to num_inter
      S2  fx[left[j]] += g(x[left[j]], x[right[j]])
      S3  fx[right[j]] += g(x[left[j]], x[right[j]])
   enddo
   do k=1 to num_nodes
      S4  vx[k] += fx[k]
   enddo
enddo

Figure 5.1: Simplified moldyn example

calculates the forces on the molecules using the left and right index arrays, which indicate interaction pairs. Previous work [33, 50] refers to left and right as the access or index arrays, and x, vx, and fx as base or data arrays.

Using the simplified moldyn benchmark in Figure 5.1, this section describes some existing run-time reordering transformations for improving the data locality within the j loop. We also describe the application of sparse tiling transformations, which improve locality between the i, j, and k loops. Finally, we present experimental results for all three benchmarks (irreg, nbf, and moldyn) on two different machines when various compositions of run-time reordering transformations are applied.

5.1.1 Run-time Data Reordering Transformations

Given a loop with non-affine memory references like the j loop in Figure 5.1, run-time data reordering transformations attempt to improve the spatial locality in the loop by reordering the data based on the order in which it is
Figure 5.2: Example mapping of iterations in the j loop to locations in the data arrays \(x\) and \(fx\). Here, circles represent iterations of the j loop inside one iteration of the outer time-stepping loop.

referred in the loop. In this example, we can use the consecutive packing (CPACK) [33] or graph partitioning (Gpart) [50] data reordering transformation to reorder the \(x\), \(fx\), and \(vx\) arrays based on the iteration to data mapping in the j loop.

CPACK works by inspecting the data mapping and generating a data reordering. Figure 5.2 shows one possible mapping of iterations in the j loop to locations in the data arrays \(x\) and \(fx\) based on values in the index arrays \texttt{left}\texttt{right}. A CPACK data reordering results in the new data mapping shown in Figure 5.3. Notice that the data has been reordered based on the memory reference order of the original mapping, therefore the computation exhibits better spatial locality.

Partitioning algorithms like Gpart [50] logically operate on a graph where each data location is a node. There is an edge between two nodes whenever their associated data is accessed within a loop iteration. By partitioning the nodes (ie. data) of the graph so that the data associated with each partition fits
Figure 5.3: Example of Figure 5.2 mapping after the CPACK data reordering into (some level of) cache and ordering the data consecutively within a partition, Gpart improves the spatial locality of the computation.

5.1.2 Run-time Iteration Reordering Transformations

Often it is beneficial for an iteration reordering of a loop with non-affine memory accesses to follow a data reordering [33, 50]. The goal of iteration reordering is to reorder the iterations based on, for instance, the location of the data being accessed in the loop.

One such iteration-reordering transformation is lexicographical grouping (lexGroup) [33]. For the simple moldyn example, Figure 5.4 shows how lexGroup further changes the data mapping between the iterations in the j loop to locations in the data arrays x and fx. Notice that iterations that touch the same or adjacent data locations now execute consecutively, therefore the computation exhibits better temporal and spatial locality.

We experimented with the iteration-reordering transformations bucket tiling [91] and lexicographical sorting [50] as well. However, lexicographical group-
Figure 5.4: Example of Figure 5.2 mapping after the CPACK data reordering followed by a lexGroup iteration reordering

ing (lexGroup) consistently exhibits the best performance to overhead trade-off on our benchmarks; therefore, the results in this chapter always use lexGroup for reordering the iterations of the j loop.

5.1.3 The Sparse Tiling Iteration Reordering Transformation

Sparse tiling programming transformations, cache blocking [35] and full sparse tiling [119], were developed for an important kernel used in Finite Element Methods, Gauss-Seidel. The sparse tiling inspector generates tiles or iteration slices [98] that cut between loops or across an outer loop and that only access a subset of the total data. By performing an iteration reordering based on a sparse tiling, locality between loops or iterations of an outer loop improves.

Sparse tiling differs from other iteration-reordering transformations in four ways.

- Sparse tiling improves the locality between loops and across iterations of outer loops even when there are data dependences. Other run-time iteration-
Figure 5.5: We highlight the iterations of one sparse tile for the code in Figure 5.1. The j loop has been blocked to provide a seed partitioning. In the full sparse-tiled executor code, the iterations within a tile are executed atomically.

reordering transformations for data locality are not applicable when there are data dependences.

- Whereas other run-time iteration reordering transformations for locality are realized with inspectors that traverse the data mappings, sparse tiling inspectors traverse the dependences.

- Sparse tiling can also be used to provide a coarser granularity of parallelism than other run-time reordering transformations for parallelism [122].

The sparse tiling inspector starts with a seed partitioning of iterations in one of the loops (or in one iteration of an outer loop). If other data and iteration-reordering transformations have been applied to the loop being partitioned, then consecutive iterations in the loop have good locality and a simple block partitioning of the iterations is sufficient to obtain an effective seed partitioning. From this seed partitioning tiles are grown to the other loops involved in the sparse tiling by a traversal of the data dependences between loops (or between iterations of
an outer loop). The main difference between cache blocking and full sparse tiling is how tile growth occurs. In cache blocking [35], the seed partitioning occurs on the first iteration of an outer loop and then tiles are grown by shrinking each partition for later iterations of that outer loop. The remaining iteration points are assigned to one tile. Full sparse tiling allows the seed partitioning to occur at any loop or iteration within an outer loop, and tiles are grown across data dependences to earlier and later loops or iterations. The tiles are grown and will be executed according to the seed partition ordering. When growing to earlier loops or iterations of an outer loop, all sources for data dependences whose target is in the current seed partition must be added to the tile, if not already assigned to a previous tile. When growing tiles to later loops or iterations of an outer loop, only targets whose sources are in the current seed partition or earlier seed partitions can be included in the current tile. This way, there are no leftover iteration points that need to be handled separately.

For the simplified moldyn example, Figure 5.5 shows the status of the data dependences between iterations of the i, j, and k loops after applying the data reordering transformation CPACK and iteration reordering transformation lexGroup. A full sparse tiling iteration reordering causes subsets of all three loops to be executed atomically as sparse tiles. Figure 5.5 highlights one such sparse tile where the j loop has been blocked to create a seed partitioning. The highlighted iterations that make up the first tile execute in the following order: iterations 4,5,2, and 6 in loop i, iterations 1,4,2, and 6 in loop j, and iterations 4 and 2 in loop k. The second tile executes the remaining iterations. Figure 5.6 shows the executor that iterates over tiles and then within the i, j, and k loops. Since iterations within all three loops touch the same or adjacent data locations, locality between the loops is improved in the new schedule.

In our experiments, we also apply a data reordering transformation, tile
do s = 1 to num_steps
    do t=1 to num_tiles
        do i4 in sched(t,1)
            x3[i4] = x3[i4] + vx3[i4] + fx3[i4]
        enddo
        do j4 in sched(t,2)
            fx3[left3[j4]] += g(x3[left3[j4]], x3[right3[j4]])
            fx3[right3[j4]] += g(x3[left3[j4]], x3[right3[j4]])
        enddo
        do k4 in sched(t,2)
            vx3[k4] += fx3[k4]
        enddo
    enddo
enddo

Figure 5.6: Sparse-tiled executor when the composed inspector performs CPACK, lexGroup, CPACK, lexGroup, full sparse tiling, and tilePack.

Figure 5.7: Tile packing reorders the iterations and corresponding data locations for the i and k loops so that the computation in each tile has better spatial locality.
<table>
<thead>
<tr>
<th>Data set</th>
<th>nodes</th>
<th>edges</th>
<th>edges/nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>foil</td>
<td>144,649</td>
<td>1,074,393</td>
<td>7.43</td>
</tr>
<tr>
<td>auto</td>
<td>448,695</td>
<td>3,314,611</td>
<td>7.39</td>
</tr>
<tr>
<td>mol1</td>
<td>131,072</td>
<td>1,179,648</td>
<td>9</td>
</tr>
<tr>
<td>mol2</td>
<td>442,368</td>
<td>3,981,312</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5.1: Data sets for experiments done with `irreg`, `nbf`, and `moldyn` benchmarks. foil and auto are used with `irreg`. mol1 and mol2 are used with `nbf` and `moldyn`.

packing (tilePack), after applying full sparse tiling. TilePack reorders the data arrays based on how data is accessed within tiles. For example, in Figure 5.7 tilePack creates the data ordering 4, 2, 5, 6, 3, 1, resulting in the consecutive ordering of data accessed within the highlighted tile.

5.1.4 Experimental Results

The sizes of the data sets we use in terms of nodes, edges, and the ratio of edges per node are given in Table 5.1.4. These datasets were made available to us by Han and Tseng, and are the same datasets they use in their Gpart experiments [53]. The foil dataset is a 3D mesh of a parafoil, and the auto is a 3D mesh of a GM Saturn automobile. The mol datasets are created by placing a number of molecules semi-uniformly in a 3D space. An edge is placed between two molecules if they are within 1.5 angstroms of each other in simulation space. The mol1 and mol2 datasets have an edge to node ratio of 9 due to the 1.5 angstrom cutoff radius. Han and Tseng experiment with larger edge to node ratios in [53].

The composed run-time reordering transformations are executed on two architectures, which are described in Table 5.2. In our experiments, we target the L1 cache when selecting parameters, such as partition size, for Gpart and full sparse tiling.

Both the baseline benchmarks and the executors for the run-time re-
Table 5.2: Descriptions of computer architectures used in composed transformation experiments

<table>
<thead>
<tr>
<th>Chip</th>
<th>MHz</th>
<th>L1 cache</th>
<th>L2 cache</th>
<th>compilation command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power3</td>
<td>375</td>
<td>64KB</td>
<td>8MB</td>
<td>xlc -bmaxdata:0x80000000 -bmaxstack:0x10000000 -O3 -DNDEBUG</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>1700</td>
<td>8KB</td>
<td>256KB</td>
<td>gcc -O2 -DNDEBUG</td>
</tr>
</tbody>
</table>

Table 5.3: The baseline execution times in seconds for each processor, program, and dataset combination. Below each dataset is the size of the dataset in the particular program.

<table>
<thead>
<tr>
<th>Machine</th>
<th>irreg</th>
<th>nbf</th>
<th>moldyn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>foil</td>
<td>mol1</td>
<td>mol1</td>
</tr>
<tr>
<td></td>
<td>auto</td>
<td>mol2</td>
<td>mol2</td>
</tr>
<tr>
<td></td>
<td>10.4MB</td>
<td>7MB</td>
<td>18MB</td>
</tr>
<tr>
<td></td>
<td>32.135MB</td>
<td>23.625MB</td>
<td>60.75MB</td>
</tr>
<tr>
<td>Power3</td>
<td>0.53</td>
<td>1.72</td>
<td>4.07</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>0.60</td>
<td>1.42</td>
<td>5.94</td>
</tr>
<tr>
<td>(1.7GHz)</td>
<td>2.64</td>
<td>9.46</td>
<td>18.34</td>
</tr>
<tr>
<td></td>
<td>2.43</td>
<td>5.38</td>
<td>20.54</td>
</tr>
</tbody>
</table>

ordering transformation compositions use inter-array data regrouping [34] to leverage shared memory reference patterns between data arrays. All compositions we consider consist of a data reordering transformation (CPACK or Gpart) followed by the iteration-reordering transformation lexicographical grouping (lexGroup) for the j loop. In some experiments, we also perform the composition CPACK, lexGroup, CPACK, lexGroup. Finally, we apply full sparse tiling (FST) after the other compositions to see if improving the locality between the i, j, and k loops results in better performance.

Table 5.3 lists the baseline execution times for all combinations of the processors, programs, and datasets. Below each dataset is the size of the dataset in the particular problem. The sizes for mol1 and mol2 differ in nbf and moldyn, because nbf stores 16 bytes of data for every node, moldyn stores 72 bytes per node, and they use different sparse matrix formats. Figures 5.8 and 5.9 show
the normalized execution times for the executor of the various compositions. The number of outermost loop iterations (time steps) required to amortize the overhead of the inspector are shown in Figures 5.10 and 5.11. We calculate this number by taking the execution time of the inspector and dividing by the savings per time step observed in the executor.

When we apply our full sparse tiling transformation in composition with other run-time data and iteration-reordering transformations, we observe mixed results on the Power3. On the Pentium 4, using full sparse tiling in composition with the other data and iteration-reordering transformations results in improved performance for all our benchmarks and data sets. The results for the moldyn benchmark are especially impressive.

The moldyn benchmark does more computation and accesses more data than the other two benchmarks. In fact, for each molecule 72 bytes of data are stored. On the Pentium 4, the cache line is only 64 bytes long. Therefore, the
Figure 5.9: Normalized execution time of executor without inspector overhead on the Pentium 4 (1.7GHz)

Figure 5.10: Amortization of the inspector overhead on the Power3 in number of outer loop iterations based on the savings per iteration experienced by the outer loop in the executor
data reordering transformations, which improve spatial locality, have less effect than iteration-reordering transformations like full sparse tiling. Full sparse tiling improves the performance in this benchmark to such an extent that it is actually easier to amortize the inspectors that include full sparse tiling (see Figure 5.11).

### 5.2 Framework Terminology

In this section, we review iteration-reordering framework terminology using the simplified moldyn code in Figure 5.1 as an example. Pugh and Wonnacott introduced the idea of using uninterpreted function symbols to statically describe the values in index arrays and other non-affine memory references [97]. Here we use such symbols to describe the data mappings and dependences in the simplified moldyn example (Figure 5.1).
5.2.1 Loops, Statements, and Data

The traditional literature on loop transformations represents each iteration within a loop nest as an integer tuple, $\vec{p} = [p_1, ..., p_n]$, where $p_q$ is the value of the iteration variable for the $q$th loop. Thus, a loop’s iteration space is a set of integer tuples with constraints indicating the loop bounds,

$$\{(p_1, ..., p_n)| lb_1 \leq p_1 \leq ub_1 \land \cdots \land lb_n \leq p_n \leq ub_n\}.$$

This representation is not very convenient for representing transformations that operate on a collection of loops that are not perfectly nested. For instance, there are three traditional iteration spaces in the code shown in figure 5.1, and it is awkward to express how the sparse tiling run-time reordering transformation operates across all three. Ahmed et al. [5] and Kelly-Pugh [73] give two different methods for building a unified iteration space. In this dissertation, we use the Kelly-Pugh method. For the simplified moldyn example in Figure 5.1, they would use a four-dimensional space. Each loop that a statement resides within corresponds to a pair of dimensions, where the first dimension of the pair is a value of the index variable, and the second dimension is the numerical order of the enclosed loop or statement. A program executes its iterations in lexicographic order of the unified iteration space.

For instance, using this representation, the $[s, k]$-th iteration of S4 is denoted $[s, 3, k, 1]$ since S4 is in the third statement (loop j) of the outer loop, and its the first statement within the k loop. The unified iteration space $I_0$ for the (untransformed) program is the following set:

$$I_0 = \{[s, 1, i, 1] | (1 \leq s \leq \text{num\_steps}) \text{ and } (1 \leq i \leq \text{num\_nodes})\}$$

$$\cup \{[s, 2, j, q] | (1 \leq s \leq \text{num\_steps}) \text{ and } (1 \leq j \leq \text{num\_inter}) \text{ and } (1 \leq q \leq 2)\}$$

$$\cup \{[s, 3, k, 1] | (1 \leq s \leq \text{num\_steps}) \text{ and } (1 \leq k \leq \text{num\_nodes})\}.$$
Next we will define data mappings and dependences for this unified iteration space.

5.2.2 Data Mappings

Each array has an associated data space represented with an integer tuple set with the same dimensionality as the array. The simplified moldyn example contains the following five data spaces:

\[
x_0 = \{ [m] \mid 1 \leq m \leq \text{num\_nodes} \},
\]

\[
v x_0 = \{ [m] \mid 1 \leq m \leq \text{num\_nodes} \},
\]

\[
f x_0 = \{ [m] \mid 1 \leq m \leq \text{num\_nodes} \},
\]

\[
\text{left}_0 = \{ [m] \mid 1 \leq m \leq \text{num\_inter} \}, \text{ and}
\]

\[
\text{right}_0 = \{ [m] \mid 1 \leq m \leq \text{num\_inter} \}.
\]

The subscripts “0” are used here since these are the data spaces for the original, untransformed program.

Define a data mapping \( M_{I \rightarrow a} \) from iterations to sets of tuples in a data space \( a \), so that for each iteration \( \bar{p} \in I \), \( M_{I \rightarrow a}(\bar{p}) \) represents the set of locations that are referenced by iteration tuple \( \bar{p} \). Notice that the subscript “\( I \rightarrow a \)” gives the domain and range of the mapping.

The moldyn example has the following data mappings:

\[
M_{I_0 \rightarrow x_0} = \{ [s, 1, i, 1] \rightarrow [i] \} \cup \{ [s, 2, j, q] \rightarrow [\text{left}(j)] \}
\]

\[
\quad \cup \{ [s, 2, j, q] \rightarrow [\text{right}(j)] \},
\]

\[
M_{I_0 \rightarrow f x_0} = \{ [s, 1, i, 1] \rightarrow [i] \} \cup \{ [s, 2, j, 1] \rightarrow [\text{left}(j)] \}
\]

\[
\quad \cup \{ [s, 2, j, 2] \rightarrow [\text{right}(j)] \} \cup \{ [s, 3, k, 1] \rightarrow [k] \},
\]

\[
M_{I_0 \rightarrow \text{left}_0} = \{ [s, 1, i, 1] \rightarrow [i] \} \cup \{ [s, 3, k, 1] \rightarrow [k] \},
\]

\[
M_{I_0 \rightarrow \text{right}_0} = \{ [s, 2, j, q] \rightarrow [j] \}, \text{ and } M_{I_0 \rightarrow \text{right}_0} = M_{I_0 \rightarrow \text{left}_0}.
\]
Here we use uninterpreted function symbols to abstractly represent the data mappings for which Figure 5.2 shows concrete examples.

### 5.2.3 Dependen ces

Define the *dependences* $D_{I-1}$ to be the set of directed edges between iterations $\bar{p} \in I$ that represent dependent computations. This set consists of the data dependence relations between Statement $SV$ and $SW$ denoted as $d_{VW}$, with $1 \leq V, W \leq 4$ in the simplified moldyn example. For example, the dependences between Statement $S1 ([s, 1, i, 1])$ and Statements $S2 ([s, 2, j, 1])$ and $S3 ([s, 2, j, 2])$ due to the $x$ and $fx$ arrays are specified with the following dependence relation:

$$ d_{12} \cup d_{13} = \{ [s, 1, i, 1] \rightarrow [s', 2, j, q] \mid (s \leq s') \text{ and } (1 \leq q \leq 2) \}
\quad \text{and} \quad (i = left(j) \lor i = right(j)) \}.$$

The dependences between Statements $S2 ([s, 2, j, 1])$ and $S3 ([s, 2, j, 2])$ and Statement $S4 ([s, 3, k, 1])$ due to the $fx$ arrays are specified with the following set of dependence relations:

$$ d_{24} \cup d_{34} = \{ [s, 2, j, q] \rightarrow [s', 3, k, 1] \mid (s \leq s') \text{ and } (1 \leq q \leq 2) \}
\quad \text{and} \quad (k = left(j) \lor k = right(j)) \}.$$

The arrows in Figure 5.5 represent concrete examples of these dependences. Notice that the dependences $d_{12} \cup d_{13}$ are symmetric to the dependences $d_{24} \cup d_{34}$ since both sets of dependences have constraints involving the $left$ and $right$ index arrays.

### 5.3 Run-time Reordering Transformations

With run-time reordering transformations, an inspector traverses data mappings or dependences transformed by reorderings produced by earlier inspec-
tors. Thus, we want our framework to describe the data mappings and dependences in effect at all stages of the transformation process. At compile-time the data and iteration reorderings are expressed with uninterpreted function symbols. At run-time the inspectors traverse and generate index arrays to store the reordering functions.

Formally, a data reordering transformation is expressed with a mapping $R_{a \rightarrow a'}$, where the data that was originally stored in location $m$ is relocated to $R_{a \rightarrow a'}(m)$. We do not need to consider the legality of a data mapping since data mappings do not affect data dependences - any one-to-one data remapping is legal. The result of remapping an array $a$ is a new data mapping, specifically

$$M_{I \rightarrow a'} = \{ \vec{p} \rightarrow R_{a \rightarrow a'}(m) | m \in M_{I \rightarrow a}(\vec{p}) \text{ and } \vec{p} \in I \}.$$  

An iteration-reordering transformation is expressed with a mapping $T_{I \rightarrow I'}$ that assigns each iteration $\vec{p}$ in iteration space $I$ to iteration $T_{I \rightarrow I'}(\vec{p})$ in a new iteration space $I'$. The new execution order is given by the lexicographic order of the iterations in $I'$. Lexicographical order on integer tuples can be defined as follows [72]: writing $(x_1, ..., x_n) < (y_1, ..., y_n)$ means there exists an $m$ such that $(\forall i : 1 \leq i < m \Rightarrow x_i = y_i)$ and $(x_m < y_m)$.

For iteration-reordering transformations, the new execution order must respect all the dependences of the original. Thus for each $\{ \vec{p} \rightarrow \vec{q} \} \in D_{I \rightarrow I}$, $T_{I \rightarrow I'}(\vec{p})$ must be lexicographically earlier than $T_{I \rightarrow I'}(\vec{q})$.

The dependences of the transformed iteration space are

$$D_{I' \rightarrow I'} = \{ T_{I \rightarrow I'}(\vec{p}) \rightarrow T_{I \rightarrow I'}(\vec{q}) | \vec{p} \rightarrow \vec{q} \in D_{I \rightarrow I} \},$$

and the new data mapping $M_{I' \rightarrow a}$ for each array $a$ is

$$M_{I' \rightarrow a} = \{ T_{I \rightarrow I'}(\vec{p}) \rightarrow M_{I \rightarrow a}(\vec{p}) | \vec{p} \in I \}.$$  

Given the new dependences and data mappings, we can plan further run-time transformations at compile time.
Within this framework run-time reordering transformations operate on subspaces within the unified iteration space. For each mapping of statements to unified iteration space a subspace can be specified by selecting a subset of dimensions in the mapping. A subspace is a candidate for run-time reordering transformations whenever the statements within the subspace involve non-affine memory references.

Data run-time reordering transformations are always legal since they do not affect dependences. Only the identity iteration reordering is possible if dependences between iterations in the subspace completely order the execution. Reduction dependences are the exception, because they allow some reordering. Statements S2 and S3 in Figure 5.1 have reduction dependences carried by the j loop on the write to $f_x$. Some run-time iteration-reordering transformations (eg. lexicographical grouping, lexicographical ordering, and bucket tiling) can only be applied when there are no dependences or only reduction dependences between iterations in the selected subspace. A partially parallel loop requires synchronization between the execution of dependent iteration subsets. When the subspace has dependences involving non-affine memory references, run-time iteration-reordering transformations such as run-time partial parallelization and sparse tiling satisfy the constraints ordained by the dependences by inspecting the dependences.

Run-time reordering transformations for partial parallelism traverse all the data dependences within an iteration subspace and create a run-time parallel schedule with maximal parallelism [103]. Such parallelism can be expressed within our framework by mapping parallel iterations to the same point in the unified iteration space. In Chapter 2, we grouped iterations into tiles and create a tile dependence graph. The tile dependence graph can be executed in parallel using a master/worker implementation that maintains the dependences between
tiles. This results in courser-grained parallelism than a schedule with maximal parallelism.

Sparse tiling transformations partition a portion of the subspace and then grow tiles that respect the data dependences throughout the rest of the subspace. Since a loop is typically the portion of the subspace initially partitioned, sparse tiles are grown across dependences between loops or between iterations of an outer loop. By mapping all independent tiles to the same tile number, parallelism between tiles can be expressed, but a master/worker implementation might result in less synchronization costs.

5.4 Composing Transformations

This section illustrate how to specify the effects of applying several run-time data and iteration-reordering transformations to the moldyn example.

5.4.1 Run-time Data Reordering

Run-time data reordering inspectors traverse data mappings and generate a data reordering function. Figure 5.12 shows the CPACK inspector code specialized for the original data mapping \( M_{t_0 \rightarrow x_0} \) (specified in section 5.2.2) in the simplified moldyn example. This specialized CPACK inspector is called by the composed inspector in Figure 5.13.

The effect of CPACK can be specified at compile time by changing all the data mappings that involve the array being reordered. In the simplified moldyn example, it makes sense to construct the same reordering for the \( x, f x, \) and \( v x \) arrays. Let \( R_{x_0 \rightarrow x_1} = \{ m \rightarrow m_1 \mid m_1 = \sigma_{cp}(m) \} \) specify the run-time data reordering on the \( x \) array, where \( x_0 \) is the data space for the \( x \) array in its original order, and \( x_1 \) is the data space for the remapped array \( x_1 \). The new data
CPACK_M_I0_to_x0(left,right)
// initialize alreadyOrdered bit vector
// to all false
count = 0
do j=1 to num_inter
    mem_loc1 = left[j]
    mem_loc2 = right[j]

    if not alreadyOrdered(mem_loc1)
        sigma_cp_inv[count] = mem_loc1
        alreadyOrdered(mem_loc1) = true
        count = count + 1
    endif

    if not alreadyOrdered(mem_loc2)
        sigma_cp_inv[count] = mem_loc2
        alreadyOrdered(mem_loc2) = true
        count = count + 1
    endif
enddo

do i=1 to num_nodes
    if not alreadyOrdered(i)
        sigma_cp_inv[count] = i
        count = count + 1
    endif
enddo

return sigma_cp_inv

Figure 5.12: First CPACK inspector for moldyn called from composed inspector in Figure 5.13
// First application of CPACK
sigma_cp_inv = CPACK_M_I0_to_x0(left,right)
sigma_cp = calcInverse(sigma_cp_inv)

// First application of lexGroup
delta_lg = lexGroup_M_I0_to_x1(left,right,sigma_cp)
delta_lg_inv = calcInverse(delta_lg)

// Second application of CPACK
sigma_cp2_inv = CPACK_M_I1_to_x1(left,right,sigma_cp,delta_lg_inv)
sigma_cp2 = calcInverse(sigma_cp2_inv)

// Second application of lexGroup
delta_lg2 = lexGroup_M_I1_to_x2(left,right,
sigma_cp,delta_lg_inv,sigma_cp2)
delta_lg2_inv = calcInverse(delta_lg2)

// Reorder data arrays to reflect final data mapping
x2 = remapArray_R_x0_to_x2(x, sigma_cp, sigma_cp2)

// Adjust values in index arrays to reflect final data mapping
left = adjustIndexArray_R_x0_to_x2(left,
sigma_cp,sigma_cp2)
right = adjustIndexArray_R_x0_to_x2(right,
sigma_cp,sigma_cp2)

// Reorder index arrays to implement final iteration reordering
left2 = remapArray_T_I0_to_I2(left, delta_lg,delta_lg2)
right2 = remapArray_T_I0_to_I2(right, delta_lg,delta_lg2)

Figure 5.13: Composed inspector for CPACK, lexGroup, CPACK, and lexGroup composition
mapping is specified as follows:

\[ M_{I_0 \rightarrow x_1} = \{[s, 1, i, 1] \rightarrow [\sigma_{cp}(i)]\} \cup \{[s, 2, j, q] \rightarrow [\sigma_{cp}(left(j))]\} \]

\[ \cup \{[s, 2, j, q] \rightarrow [\sigma_{cp}(right(j))]\}. \]

A data reordering \( \sigma_{gp} \) based on Gpart orders data within the same partition consecutively. In the simple moldyn example, the abstract specification of the iteration to data mappings after applying Gpart is obtained by replacing \( \sigma_{cp} \) with \( \sigma_{gp} \).

**5.4.2 Run-time Iteration Reordering**

If an iteration-reordering transformation on the \( j \) loop follows a data reordering transformation (as in our experiments), then the inspector for the iteration-reordering transformation will traverse the updated data mappings. In the simplified moldyn example, lexGroup will iterate over the data mappings that include the \( \sigma_{cp} \) function if lexGroup is performed after CPACK. The iteration reordering of the \( i, j, \) and \( k \) loops is specified as follows:

\[ T_{I_0 \rightarrow I_1} = \{[s, 1, i, 1] \rightarrow [s, 1, i_1, 1] \mid i_1 = \sigma_{cp}(i)\} \]

\[ \cup \{[s, 2, j, q] \rightarrow [s, 2, j_1, q] \mid j_1 = \delta_{lg}(j)\} \]

\[ \cup \{[s, 3, k, q] \rightarrow [s, 3, k_1, 1] \mid k_1 = \sigma_{cp}(k)\}. \]

Since each iteration of the \( i \) and \( k \) loops directly maps to the \( x, fx, \) and \( vx \) arrays, the data reordering function generated for the arrays, \( \sigma_{cp} \), is used for reordering the \( i \) and \( k \) loops as well. The transformation \( T_{I_0 \rightarrow I_1} \) is legal because the only loop-carried dependences within the \( i, j, \) or \( k \) loops are reduction dependences between iterations of the \( j \) loop.

Due to the iteration reordering, the data mappings become

\[ M_{I_1 \rightarrow x_1} = \{[s, 1, \sigma_{cp}(i), 1] \rightarrow [\sigma_{cp}(i)]\} \cup \{[s, 2, \delta_{lg}(j), q] \rightarrow [\sigma_{cp}(left(j))]\} \]

\[ \cup \{[s, 2, \delta_{lg}(j), q] \rightarrow [\sigma_{cp}(right(j))]\}, \]
Figure 5.14: Second CPACK inspector for moldyn called from composed inspector in Figure 5.13

and the dependences \((d'_{12} \cup d'_{13} \cup d'_{24} \cup d'_{34}) \subset D_{I_1 \rightarrow I_1}\), become

\[
d'_{12} \cup d'_{13} = \{[s, 1, \sigma_{cp}(i), 1] \rightarrow [s', 2, \delta_{lg}(j), q] | (s \leq s') \text{ and } (1 \leq q \leq 2) \\
\text{and } (i = \text{left}(j) \text{ or } i = \text{right}(j))\}
\]

and \(d'_{24} \cup d'_{34} = \{[s, 2, \delta_{lg}(j), q] \rightarrow [s', 3, \sigma_{cp}(k), 1] | (s \leq s') \text{ and } (1 \leq q \leq 2) \\
\text{and } (k = \text{left}(j) \text{ or } k = \text{right}(j))\}\).

### 5.4.3 Subsequent Transformations

When composing run-time reordering transformations, specialized instances of the relevant inspectors can be created that account for changes to the data mappings and dependences incurred by any previously planned inspectors. The explicit abstract description of how run-time reordering transformations affect each other allows new run-time reordering transformation compositions. For example, it is possible to generate another CPACK data reordering and lexGroup iteration reordering after generating the first CPACK and lexGroup reorderings.
do s = 1 to num_steps
    do i2 = 1 to num_nodes
        x2[i2] = x2[i2] + vx2[i2] + fx2[i2]
    enddo

do j2 = 1 to num_inter
    fx2[left2[j2]] += g(x2[left2[j2]], x2[right2[j2]])
    fx2[right2[j2]] += g(x2[left2[j2]], x2[right2[j2]])
enddo

do k2 = 1 to num_nodes
    vx2[k2] += fx2[k2]
enddo
enddo

Figure 5.15: Executor for simple moldyn example when inspector applies CPACK, lexGroup, CPACK, lexGroup composition

Figure 5.14 shows how the second CPACK inspector is specialized to traverse the data mappings resulting from the first data and iteration-reordering functions, $M_{I_1 \to x_1}$. The array \texttt{delta\_lg\_inv} stores the inverse of the iteration reordering function $\delta_{lg}$. The compile-time specification for the second CPACK inspector is

$$R_{x_1 \to x_2} = \{m_1 \to m_2 \mid m_2 = \sigma_{cp2}(m_1)\}.$$ 

A second iteration-reordering transformation for loop $j$ traverses the data mapping $M_{I_1 \to x_2}$ and generates the reordering function $\delta_{lg2}$ to implement the transformation $T_{I_1 \to I_2}$, where $M_{I_1 \to x_2}$ and $T_{I_1 \to I_2}$ are specified as follows:

$$M_{I_1 \to x_2} = \{[s, 1, \sigma_{cp}(i), 1] \to [\sigma_{cp2}(\sigma_{cp}(i)))]\}$$

$$\cup \{[s, 2, \delta_{lg}(j), q] \to [\sigma_{cp2}(\sigma_{cp}(left(j)))]\}$$

$$\cup \{[s, 2, \delta_{lg}(j), q] \to [\sigma_{cp2}(\sigma_{cp}(right(j)))]\}$$

and

$$T_{I_1 \to I_2} = \{[s, 1, i_1, 1] \to [s, 1, i_2, 1] \mid i_2 = \sigma_{cp2}(i_1)\}$$
\[
\cup \{[s, 2, j_1, q] \rightarrow [s, 2, j_2, q] \mid j_2 = \delta_{j_2}(j_1)\} \\
\cup \{[s, 3, k_1, 1] \rightarrow [s, 3, k_2, 1] \mid k_2 = \sigma_{k_2}(k_1)\}.
\]

With a compile-time description of the effects of a run-time data or iteration reordering, it is possible to plan compositions of run-time transformations and generate code for the composed data remappings at the end of all inspection. This is done by manipulating reordering function arrays (\texttt{sigma.cp}, \texttt{delta.lg}, etc.) at runtime. The composed \textit{inspector} in Figure 5.13 remaps and updates the data and index arrays accordingly after all data and iteration reorderings have been computed. Figure 5.15 shows the composed \textit{executor}.

### 5.4.4 Sparse Tiling

As iteration-reordering transformations, sparse tiling transformations can also be composed with other run-time reordering transformations. The main difference between sparse tiling transformations and other run-time reordering transformations for locality is that sparse tiling is applicable within subspaces of the unified iteration space that contain data dependences. This is possible because sparse tiling inspectors traverse the data dependences and choose reorderings that respect the data dependences.

For the simplified \texttt{moldyn} example, applying sparse tiling after the CPACK, lexGroup, CPACK, lexGroup series of run-time transformations described in Section 5.4.3 can be specified with the following mapping:

\[
T_{l_2 \rightarrow l_3} = \{[s, 1, i_2, 1] \rightarrow [s, 1, \theta(1, i_3), 1, i_3, 1] \mid i_3 = i_2\} \\
\cup \{[s, 2, j_2, q] \rightarrow [s, 1, \theta(2, j_3), 2, j_3, q] \mid j_3 = j_2\} \\
\cup \{[s, 3, k_2, 1] \rightarrow [s, 1, \theta(3, k_3), 3, k_3, 1] \mid k_3 = k_2\}.
\]

The tiling function \(\theta\) assigns a subspace of the unified iteration space to tile numbers. The subspace being sparse tiled in this example is \([1, i_2] \cup [2, j_2] \cup \ldots\).
\[ [3, k_2] \], with the seed partitioning occurring on the \([2, j_2]\) portion of the subspace. Figure 5.5 illustrates an instance of sparse-tiled \texttt{moldyn} that uses full sparse tiling for tile growth.

In Figure 5.15, the computation exhibits better spatial locality if the data arrays are remapped after sparse tiling as shown in Figure 5.7. Specifically if the data item (and corresponding iteration) numbered as 6 is put before 3, and 2 before 5, there is better locality. We refer to reordering the data and iterations in loops \(i\) and \(k\) based on the tiling function as \textit{tile packing} (tilePack). TilePack uses an inspector that traverses the tiling function to generate the following data and iteration-reordering transformations:

\[
R_{x_2 \rightarrow x_3} = \{[m_2] \rightarrow [m_3] \mid m_3 = \sigma_{tp}(m_2)\}
\]

and

\[
T_{I_3 \rightarrow I_4} = \{[s, 1, t, 1, i_3, 1] \rightarrow [s, 1, t, 1, i_4, 1] \mid i_4 = \sigma_{tp}(i_3)\}
\]

\[
\cup \{[s, 1, t, 2, j_3, q] \rightarrow [s, 1, t, 2, j_4, q] \mid j_4 = j_3\}
\]

\[
\cup \{[s, 1, t, 3, k_3, 1] \rightarrow [s, 1, t, 3, k_4, 1] \mid k_4 = \sigma_{tp}(k_3)\}\}
\]

Figure 5.6 shows the executor for the simple \texttt{moldyn} example when the iteration-reordering composition \(T_{I_0 \rightarrow I_4}\) and the data reordering compositions \(R_{x_0 \rightarrow x_3}\) are generated by composing the transformation mappings discussed in this section and previous sections. Since the executor must traverse the final iteration space in lexicographical order, a schedule (indexed by the tile and all but the last dimension of the subspace being sparse tiled) is created to indicate the subset of iterations within each tile. In the running example, the schedule data structure is created as follows:

\[
sched(t, 1) = \{[i_4] \mid i_4 = \sigma_{tp}(i_3) \text{ and } \theta(1, i_3) = t\}
\]

\[
sched(t, 2) = \{[j_4] \mid j_4 = j_3 \text{ and } \theta(2, j_3) = t\}
\]

\[
sched(t, 3) = \{[k_4] \mid k_4 = \sigma_{tp}(k_3) \text{ and } \theta(3, k_3) = t\}\}.\]
5.5 Reducing the Inspector Overhead

The overhead of executing any inspector must be amortized to make run-time reordering transformations beneficial. In our experiments, we take advantage of the framework in two ways to generate efficient inspectors. First, in the benchmarks there are cases where two sets of data dependences satisfy the same constraints. Therefore, the full sparse tiling inspector need only traverse one set of data dependences while generating a legal tile function. Second, our experimental results indicate that remapping the data arrays after all run-time reordering functions have been generated reduces the execution time of inspectors that perform more than one data reordering.

Whenever two sets of data dependences satisfy the same constraints, it is only necessary to traverse one set at run-time. In the simplified moldyn example, the data dependences between Statement $S_1$ and the statements in the $j$ loop, $S_2$ and $S_3$, are symmetric to the dependences between the statements in the $j$ loop and Statement $S_4$. Therefore, our full sparse tiling inspector need only traverse one set of these dependences to grow the tiles from a seed partitioning of the $j$ loop to the $i$ and $k$ loops. A similar situation occurs in all the benchmarks we use for experiments.

The framework allows the compiler to choose when to remap a data array. Figure 5.13 illustrates a composed inspector that performs data remapping and index array adjustments $^1$ after all reordering functions are generated, and Figure 5.16 shows an inspector performing the same composition of transformations, but remapping and adjusting after each reordering function is generated. Notice that many of the functions, like $\text{CPACK}_M_{I1 \to x1_B}$, take fewer parameters in Figure 5.16 than in Figure 5.13. Since the index arrays left and right are remapped and adjusted after every transformation in Figure 5.16, the index

$^1$Adjusting the index array is the same as pointer update [33]
// First application of CPACK
sigma_cp_inv = CPACK_M_I0_to_x0(left,right)
sigma_cp = calcInverse(sigma_cp_inv)
// Reorder data arrays to reflect data mapping
x1 = remapArray_R_x0_to_x1(x,sigma_cp)
// Adjust values in index arrays
left = adjustIndexArray_R_x0_to_x1(left,sigma_cp)
right = adjustIndexArray_R_x0_to_x1(right,sigma_cp)

// First application of lexGroup
delta_lg = lexGroup_M_I0_to_x1_B(left,right)
// Reorder index arrays to implement lexGroup
left1 = remapArray_T_I0_to_I1(left, delta_lg)
right1 = remapArray_T_I0_to_I1(right, delta_lg)

// Second application of CPACK
sigma_cp2_inv = CPACK_M_I1_to_x1_B(left1,right1)
sigma_cp2 = calcInverse(sigma_cp2_inv)
// Reorder data arrays to reflect data mapping
x2 = remapArray_R_x1_to_x2(x1,sigma_cp2)
// Adjust values in index arrays
left1 = adjustIndexArray_R_x1_to_x2(left1, sigma_cp2)
right1 = adjustIndexArray_R_x1_to_x2(right1, sigma_cp2)

// Second application of lexGroup
delta_lg2 = lexGroup_M_I1_to_x2_B(left1,right1)
// Reorder index arrays to implement last iteration reordering
left2 = remapArray_T_I1_to_I2(left1,delta_lg2)
right2 = remapArray_T_I1_to_I2(right1,delta_lg2)

Figure 5.16: Composed inspector for CPACK, lexGroup, CPACK, and lexGroup where data remapping and index array updates are done immediately after the relevant reordering function is generated.
Figure 5.17: Percent reduction in inspector overhead for compositions with two or more data reorderings when data is remapped once. nbf does not benefit from the tilePack data reordering transformation; therefore, most of the compositions involving nbf do not use two or more data reordering transformations.

arrays that maintain the reordering functions are not needed. This results in fewer indirect memory references in the composed inspector and can have an effect on its performance.

Our experience suggests that remapping and adjusting the index arrays after each transformation and remapping the data arrays after all data reordering transformations leads to the most efficient inspectors. Figure 5.17 shows the percentage execution time reduction when the data arrays are remapped after all transformations as compared to remapping them after each transformation. Optimal generation of composed inspectors is an open question. Our framework allows the expression of diverse possibilities.
5.6 Related Work

Run-time reordering transformations differ from dynamic compilation transformations such as those described in [45], because reordering transformations do not change the code at runtime. Instead the code has already been transformed and inspectors create data and iteration-reordering functions, which are stored in index arrays. Both must amortize run-time overhead.

Researchers have developed run-time data dependence analysis to handle non-affine memory references [97, 109]. In [97] constraints for disproving dependences are evaluated at runtime. Rus et al. [109] take this further adding the ability to traverse all data dependences at runtime if necessary. They perform a hybrid (static and dynamic) data dependence analysis inter-procedurally. As we have described in this paper, traversing data dependences at run-time is necessary for some run-time reordering transformations.

Many run-time data reordering transformations [28, 7, 91, 33, 50] fit within our framework. Space filling curves and register tiling for sparse matrix vector multiply are two types of data reordering transformations that are more specialized. Data reorderings generated from space-filling curves [115, 87] traverse data mappings and mappings of data to spatial coordinates. The programmer must specify how data maps to spatial coordinates, therefore, such data reorderings can not be fully automated. Im and Yelick [65] have developed the SPARSITY code generator that improves the locality for the \(\vec{x}\) and \(\vec{b}\) vectors in the sparse matrix-vector multiplication \(A\vec{x} = \vec{b}\). The dynamic register blocking techniques are useful for many application domains that use sparse matrices, but the work focuses on a single algorithm.

There has been a definite progression toward complete automation of run-time reordering transformations. Initially such transformations were incorporated into applications manually for parallelism [30]. Next, libraries with run-
time transformation primitives were developed so that a programmer or compiler could insert calls to such primitives [31]. Currently, there are many run-time reordering transformations for which a compiler can automatically analyze and generate the inspectors [103, 33, 91, 50]. However, each transformation or composition of transformations are treated separately. Our framework provides a uniform representation for these transformations and describes how to compose any number of them at compile time.

5.7 Summary

This chapter motivates compositions of run-time data-reordering and iteration-reordering transformations with experimental results showing significant performance improvements for the irreg, nbf, and moldyn benchmarks. We show how to use an existing compile-time framework to formally express the changes in dependences and data mappings that occur when a composition of data and iteration-reordering transformations are performed. Representing the abstract effect of run-time data and iteration-reordering transformations at compile time is an important step toward the automatic generation of specialized inspectors. By showing that sparse tiling can be represented in our framework, we demonstrate its general applicability to other irregular codes; until the development of this framework, it was only used with Gauss-Seidel. We also use two different optimizations to improve the performance of our composed inspectors, thus reducing the overhead of composed run-time reordering transformations.

The text of this chapter is in part a reprint of the material as it appears in the proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). The dissertation author was the primary researcher and author and the co-authors listed on this publication supervised the research that forms the basis for this chapter.
Chapter 6

Conclusions and Future Work

The goal of this dissertation has been to develop run-time reordering transformations for the performance improvement of irregular applications. We developed the full sparse tiling run-time reordering transformation and presented experimental results showing that full sparse-tiled Gauss-Seidel exhibits improved performance over a typical Gauss-Seidel schedule due to data locality improvements and to the exposure of parallelism. We then used a sparse-tiled Gauss-Seidel inspector and executor within the context of a multigrid computation and as a preconditioner for a Krylov method. These implementations in the software packages FEtk [60] and PETSc [11] were used to explore the domain-specific issues involved in using sparse-tiled Gauss-Seidel with symmetric multigrid and software implementation issues involved with incorporating such run-time reordering transformations into existing software packages. Finally, we presented a framework for the compile-time composition of all run-time reordering transformations. Our results show that composing run-time reordering transformations improves the execution time in a partial differential equation benchmark and two molecular dynamics benchmarks.

This work can be extended in many directions. A few of these directions include:
Automation of dependence analysis needed to create data mappings and dependences. Since the data mappings and dependence relations are traversed at runtime within the inspector, it is essential that the number of dependences traversed are minimal. In the past, data dependence analysis has focused on answering the dependence question at compile time, when the memory references involved in a dependence are affine. By building on a combination of Nonlinear Array Dependence Analysis [97], which introduces the concept of using uninterpreted function symbols for non-affine memory references, and Hybrid Analysis [109], a framework for combining compile-time and run-time dependence analysis, we would like to investigate further simplifications of data mappings and data dependence relations through value-based dependence analysis [100].

Automatic generation of specialized and optimized inspectors and executors. Specializing an inspector for a reordering transformation in the context of its position in a composed inspector should result in less overhead than an inspector implemented in a run-time library, since the latter must be generally applicable. The need for specialized inspectors has been described in work for data locality [87] and parallelism [47]. The inspectors we created for the \texttt{irreg}, \texttt{nbf}, and \texttt{moldyn} benchmarks in Chapter 5 were generated by hand and utilize optimization techniques such as traversing only one set of similar data dependences, and remapping data only once even if multiple data reorderings are generated. Automating the code generation for the inspector and executor can leverage the work in [33], which describes compiler support for dynamic data packing, and the work in [73], which generates optimized code for compile-time transformations. Specifically, the techniques described in [73] can be used to generate the transformed executor code and the inspector code that traverses the data mappings and dependences. One extension that is necessary is the ability to symbolically project index variables that are inputs to uninterpreted function symbols. An-
other optimization possibility is to parallelize the inspector [104].

Checking the legality of run-time reordering transformations automatically at compile-time. In Chapter 2, we proved the legality of the tiling function and data reordering function generated by the full sparse tiling inspector for Gauss-Seidel. It should be possible to develop methods for performing such proofs with automatic theorem provers.

Performance modeling of run-time reordering transformation compositions to enable compile-time and run-time transformation guidance. For the complete automatic application of run-time reordering transformations, the compiler and run-time decision code must make choices between possible transformation compositions. To do this, it will be necessary to understand the effect on performance of transformation parameters, such as the size of the seed partitioning used for full sparse tiling. In the domain of data and iteration reordering, the work in [92] and [136] proposes methods for guidance when some information such as the data access pattern is not available until runtime.

Extend the full sparse tiling executor for distributed memory parallelism. This will require calculating the data footprint of all the tiles and creating an allocation of tiles to processors that results in parallel efficiency and minimal communication.

In conclusion, run-time reordering transformations composed at compile time are a promising approach for improving the performance of irregular applications. A framework for describing all run-time reordering transformations and their compile-time composition opens many possibilities for extending the impact of run-time reordering transformations.
Bibliography


15th Annual ACM SIGPLAN Symposium on Principles of Programming

code. In Proceedings of the Scientific Computing in Object-Oriented Para-
allel Environments First International Conference (ISCOPE), pages 249–256.


[69] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irreg-
ular graphs. Journal of Parallel and Distributed Computing, 48(1):96–129,

[70] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonna-
cott. New user interface for Petit and other extensions, December 1996.

[71] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonna-
cott. The omega calculator and library, version 1.1.0, November 1996.

[72] W. Kelly and W. Pugh. Finding legal reordering transformations using map-
editors, Proceedings of the 7th International Workshop on Languages and
Compilers for Parallel Computing, volume 892 of Lecture Notes in Com-

[73] W. Kelly and W. Pugh. A unifying framework for iteration reordering
transformations. Technical Report CS-TR-3430, University of Maryland,

Murphy. Open implementation design guidelines. In Proceedings of the

European Conference on Object-Oriented Programming (ECOOP).

In Proceedings of the 1997 ACM SIGPLAN Conference on Programming


