DISSERTATION

GENERALIZED FULL SPARSE TILING OF LOOP CHAINS

Submitted by

Christopher D. Krieger

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2013

Doctoral Committee:

Advisor: Michelle Mills Strout

Wim Böhm
Sanjay Radjopadhye
Jennifer Mueller

ABSTRACT

GENERALIZED FULL SPARSE TILING OF LOOP CHAINS

Computer and computational scientists are tackling increasingly large and complex problems and are seeking ways of improving the performance of their codes. The key issue faced is how to reach an effective balance between parallelism and locality. In trying to reach this balance, a problem commonly encountered is that of ascertaining the data dependences. Approaches that rely on automatic extraction of data dependences are frequently stymied by complications such as interprocedural and alias analysis. Placing the dependence analysis burden upon the programmer creates a significant barrier to adoption.

In this work, we present a new programming abstraction, the *loop chain*, that specifies a series of loops and the data they access. Given this abstraction, a compiler, inspector, or runtime optimizer can avoid the computationally expensive process of formally determining data dependences, yet still determine beneficial and legal data and iteration reorderings.

One optimization method that has been previously applied to irregular scientific codes is *full sparse tiling*. Full sparse tiling has been used to improve the performance of a handful of scientific codes, but in each case the technique had to be applied from scratch by an expert after careful manual analysis of the possible data dependence patterns. The full sparse tiling approach was extended and generalized as part of this work to apply to any code represented by the loop chain abstraction. Using only the abstraction, the generalized algorithm can produce a new data and iteration ordering as well as a parallel execution schedule.

Insight into tuning a generalized full sparse tiled application was gained through a study of the different factors influencing tile count. This work lays the foundation for an efficient autotuning approach to optimizing tile count.

ACKNOWLEDGEMENTS

I gratefully acknowledge support for this research from the Department of Energy CACHE Institute grant DE-SC04030, DOE grant DE-SC0003956, and NSF grant CCF 0746693. This research utilized the CSU ISTeC Cray HPC System supported by NSF Grant CNS-0923386. I also acknowledge the use of the machines in Intel's Manycore Testing Lab.

I also appreciate the feedback and support received throughout my graduate career from my fellow students Andy Stone, Chris Wilcox, Tomofumi Yuki, and Alan LaMielle, as well as from faculty members Cathie Olschanowsky, Sanjay Rajopadhye, and Wim Böhm.

I thank Samantha Wood for her work on full sparse tiling the matrix powers kernel and her invaluable SMOReS thesis, which was frequently consulted when analyzing the behavior of different sparse matrices.

I am also thankful for the patience and understanding of my colleagues at Intel, Ram Srinivasan, Jim Callister, Stephanie Postal, Richard Blumburg, Alex Settle, Lambert Schaelicke, Eric Borch, and Derek Cho. Their unflagging support contributed significantly to my ability to work on my research while continuing my career.

While developing the loop chain abstraction, I received valuable feedback and code examples from Xinfang Gao and Stephen Guzak of the Colorado State University Mechanical Engineering Department and from Brian Van Straalen and Sam Williams of Lawrence Berkeley National Lab.

I appreciate the help of my collaborators Paul HJ Kelly, Doru Bercea, Fabio Luporini, and Graham Markall at Imperial College of London, Carlo Bertolli at IBM Research, and Gihan Mudalige at Oxford College. Our discussions on full sparse tiling unearthed many weaknesses in early versions of the generalized full sparse tiling algorithm. I also thank them for their support regarding the airfoil benchmark.

DEDICATION

I give special thanks to my advisor, Michelle Strout, who was willing to advise, against her better judgement, a student attempting to live the oxymoron"part time PhD student." Her years of guidance, teaching, and patience with my non-standard and protracted graduate program is much appreciated.

I dedicate this work to my wife Lockey and our four sons Ian, Thomas, Jacob, and Eli. The original dedication in my master's thesis read,"To Lockey, who kept this a thesis and not a dissertation," reflecting her desire to be done with graduate school. Now, after more than a decade and over six more years of schooling, she finally has her wish.

TABLE OF CONTENTS

Chapter 1

Introduction

Computer and computational scientists are tackling increasingly large and complex problems and are seeking ways of improving the performance of their codes. One of the key issues faced by these scientists and others in the high performance computing community is how to reach an effective balance between parallelism and locality. The general solution is to bundle together work that uses the same data, thereby improving locality. That work then executes according to a parallel schedule that respects all data dependences. Many different solutions, each with its advantages and disadvantages, have been proposed but the issue remains an area of active research.

A challenge commonly encountered by existing approaches is how to ascertain the data dependences. Approaches that rely on automatic extraction of data dependences are frequently stymied by complications such as interprocedural program analysis and alias analysis. Placing the dependence analysis burden upon the programmer creates a significant barrier to adoption.

In this work, we present a new programming abstraction, *loop chains*, that allows programmers or tools to specify a series of loops and the data they access. Declaring what data is accessed by a loop is far simpler than determining data dependences. Working from this abstraction, a generalized version of the full sparse tiling approach is then able to create high locality, parallel schedules.

1.1 Balancing Parallelism and Locality in Scientific Codes

There is a significant, established code base in the scientific computing community. To date, many of these codes have been parallelized. However, these parallel codes are now

encountering scalability issues due to poor data locality, inefficient data distributions, and/or load imbalance as they are run on larger and larger systems. Code and algorithm changes are therefore necessary in order to effectively utilize new computational resources, many of which require ever increasing levels of parallelism for efficient execution.

With the increase in the amount of computational power has come an increased need for memory bandwidth to supply those resources with data. However, memory technology has not kept pace with advances in computation rate. This has led to an imbalance between the compute capabilities of multicore systems and accelerators and the ability of the memory subsystem or interconnect network to deliver data. To address this problem, codes are being modified to reduce their bandwidth requirements, often by improving data locality. This often requires changes to data and work distribution methodologies.

Given this shifting landscape of increasing compute power, often from sources such as accelerators, diminishing relative performance of memory systems, and large amounts of critical, validated scientific codes with rigid explicit data distributions yielding non-portable performance, it is no surprise that balancing parallelism and locality remains a significant challenge for the scientific computing community.

1.2 Existing Approaches to Balancing Parallelism and Locality

Various solutions have emerged for addressing the challenge of achieving and balancing both parallelism and data locality. One current approach is the incremental addition of parallelism using OpenMP [23] parallel loop pragmas. In this approach, the compiler adds the necessary code for parallel execution and synchronization. This method enables the unobtrusive addition of parallelism. Unfortunately, this approach typically provides the programmer with little control over how data and computation are grouped or distributed. For example, OpenMP only allows statically or dynamically sized blocks of contiguous iterations to be assigned to threads. Good data locality is only obtained if consecutive iterations of a loop access data in common or can be made to do so. OpenMP also does not provide a way to

group iterations of different loops together. These limitations make it difficult to improve data locality during parallel execution.

An earlier and still quite popular approach to parallelization that is significantly less incremental is to use explicit message passing libraries such as MPI [33]. MPI has been used to parallelize a significant fraction of mature parallel scientific codes. MPI development is disruptive and requires a considerable coding effort. However, the difficult aspects of the MPI programming model also lead to advantages in terms of data locality. Specifically, the programmer must manage the distribution for both the data and the computation and specify communication between nodes with explicit sends and receives. The programmer can therefore put computation together that accesses similar data.

Both OpenMP and MPI programming models have led to codes with one parallel loop after another in sequence. Communication among processing elements typically occurs between loops. At its simplest, this bulk model can be synchronized using barriers. Unfortunately, this approach can be inefficient if different processing elements take different amounts of time to complete their assigned work. This is usually the case when work cannot be evenly divided between processors or if execution times differ due to varying data access times. In this case, processors sit idle, waiting until the last processor completes its work and reaches the barrier.

In order to achieve the high levels of parallelism required to efficiently execute on large parallel machines, it is necessary to move from this bulk parallelism to asynchronous parallelism. With asynchronous parallelism, the ordering dependences between tasks are determined either in advance or dynamically at runtime. A task is allowed to execute immediately upon the completion of all its predecessor tasks. This model circumvents many of the load balancing issues of bulk synchronization. Models such as the OpenMP 3.0 task model [30], the Concurrent Collections model [17], and StarPU [3] directly expose asynchronous tasks.

Other approaches to managing the parallelism and data locality tradeoff include Partitioned Global Address Space (PGAS) languages such as Unified Parallel C [31] or Co-Array Fortran [47]. In these languages, users explicitly distribute data to nodes and can therefore

achieve good locality, just as was done using direct MPI calls to distribute data. These languages have the advantage that the necessary communication code is abstracted away behind a one-sided communication model. In this model, accessing remote data is coded in a way that is syntactically similar to local accesses and a compiler inserts the necessary data transfers. Consequently, while data distribution is still explicitly under programmer control, communication is not as cumbersome in a PGAS language. Unfortunately, PGAS languages have no language features to improve data locality across loops within a node.

Relatively new parallel programming languages such as Chapel [15] or X10 [18] provide mechanisms for specifying data locality such as places and locales, but these specifications are done on a loop by loop or computation by computation basis. The parallel constructs are considered in isolation with respect to other adjacent or surrounding constructs. There is no way to aggregate parallel loops. There is also a prohibitive cost in porting existing code to an entirely new language.

At the opposite end of the spectrum lie autoparallelization [4, 32] approaches. Under these schemes, a compiler or other tool determines the data and computation distribution automatically. Examples such as the Build To Order BLAS compiler perform data locality and parallelization optimizations across function calls [9]. An autoparallelization approach can achieve good performance portably and frees the programmer from having to make fixed choices, but also means the programmer has little ability to alter or further optimize the distributions. More significantly, automatic approaches typically rely on extensive data dependence analysis. Precise data dependence analysis is an open research challenge, fraught with difficulties such as pointer aliasing and interprocedural analysis that historically have forced these algorithms to make overly conservative assumptions on real-world codes.

1.3 Introduction to Loop Chains

Given the extremes of user-laborious, non-portable parallelization approaches or limited automatic methods previously outlined, it seems that the best answer lies between the two. Ideally, the scheduling and placement of computation and distribution of data would be

handled automatically by a compiler and/or runtime that is aware of the target system and can optimize for it. However, it is important for such a system not to rely heavily on the automatic detection of data dependences, especially across the many procedure calls that occur in modular scientific computing software.

One way to eliminate the need for automatic dependence analysis is to require the programmer to provide this information. Unfortunately, in general, and particularly in irregular codes, the programmer does not reason about the code in terms of formal data dependences between loop iterations. Even more rarely does the programmer consider dependences across different loops. What is needed is a way for the data dependences to be expressed in a way that is easily understood by a programmer and fits with his or her intuition about the code.

In this work, we introduce a new abstraction called a *loop chain* in which a sequence of parallel and/or reduction loops that explicitly share data are grouped together into a chain. A description of the data accessed within each loop is provided as part of the abstraction. The loop chain stands at an intermediate point, still close enough to the concrete code that the programmer can reason about it, yet also close enough to the explicit data dependences that the optimizer can derive ordering constraints from it.

The definition of a loop chain consists of three pieces, namely:

- Well defined iteration spaces for each loop, $\{L_0, L_2, \ldots, L_{N-1}\}$
- Data spaces for accessed data, $\{D_0, D_2, \ldots, D_{M-1}\}$
- Access relations for data read/written by each iteration of each loop:
  $R_{L_l \to D_d}(\vec{i})$ or $W_{L_l \to D_d}(\vec{i})$

Note that because the loop chain abstraction requires the explicit declaration of data access relations, it is not dependent on automatic techniques or interprocedural program analysis to discover data dependences. Because the iteration spaces are well defined, every iteration of every loop is clearly defined in the loop chain. Satisfying producer-consumer relationships for data imposes a partial ordering on the execution sequence of the iterations.

```
1  for (int t=0; t < numIters; t += 2) {
2    for (int i=0; i<numrows; i++) {
3      double diag = 1.0;
4      Ueven[i] = 0.0;
5      for (int p=IA[i]; p < IA[i+1; p++) {
6        int j = JA[p];
7        if (j==i) { diag = A[p]; }
8        else { Ueven[i] += A[p] * Uodd[j]; }
9      }
10     Ueven[i] = (F[i] - Ueven[i]) / diag;
11   }
12
13   for (int i=0; i<numrows; i++) {
14     double diag = 1.0;
15     Uodd[i] = 0.0;
16     for (int p=IA[i]; p < IA[i+1]; p++) {
17       int j = JA[p];
18       if (j==i) { diag = A[p]; }
19       else { Uodd[i] += A[p] * Ueven[j]; }
20     }
21     Uodd[i] = (F[i] - Uodd[i]) / diag;
22   }
23 }
```

Figure 1.1: The kernel of the sparse Jacobi solver for CSR sparse matrices. Output vectors Ueven and Uodd are used in an alternating, ping-pong, fashion. The loops beginning on lines 2 and 13 can be chained.

Therefore, once specified, a loop chain can be used to derive a set of loop iterations under a partial ordering. This partially ordered set of iterations makes scheduling and determining data distributions across loops possible for a compiler and/or run-time system. The flexibility of being able to schedule across loops enables better management of the data locality and parallelism tradeoff.

The concept of a loop chain can be illustrated with a concrete example. Figure 1.1 shows the C code for a sparse Jacobi solver. Given a sparse matrix A, and a vector $\vec{f}$, related by $A\vec{u} = \vec{f}$, each iteration of the sparse Jacobi method produces an approximation to the unknown vector $\vec{u}$. The recurrence equation closely resembles matrix vector multiplication and is representative of a broad class of sparse linear algebra applications. This code uses a ping-pong approach to the output vector, switching between Ueven and Uodd. The outermost loop repeats the kernel for a fixed number of repetitions.

If the two loops shown on lines 2 and 13 are combined into a loop chain, they form a chained iteration space like that shown in Figure 1.2. The specific space shown is for a sample matrix with seven rows and columns. In Figure 1.2, the boxes represent data spaces or arrays

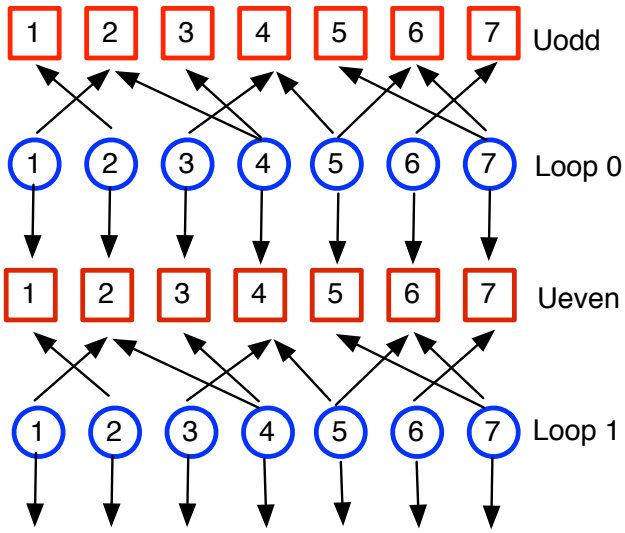


Figure 1.2: The data and iteration space view of the Jacobi code in Figure 1.1. The boxes represent the Ueven and Uodd vectors, the circles represent iterations of the two loops. The data access functions are represented by arrows. The upward pointing read arrows are irregular and dictated by the particular sparse matrix, while the downward pointing write accesses are an identity relation. Chain invariant data, such as F, IA, JA, and A, are present in the loop chain data structures but are omitted from this diagram.

D_0=Ueven and D_1=Uodd, while the circles represent iterations of the loops beginning on lines 2 and 13. The sparse matrix used as input is

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 4 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 2 & 0 & 0 & 0 \\ 0 & 9 & 2 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 2 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 1 & 4 & 7 \end{bmatrix}. \tag{1.1}$$

The iteration spaces shown for the two loops are $L_0 = \{1, 2, \ldots, 7\}$ and $L_1 = \{1, 2, \ldots, 7\}$, which is the case for the given sparse matrix with 7 rows. Arrows represent the data access relations, with upward pointing arrows indicating the data reads $R_{L_0 \to D_1} = R_{L_1 \to D_2} = \{[1] \to [2], [2] \to [1], [3] \to [4], [4] \to [2], [4] \to [3], [5] \to [4], [5] \to [6], [6] \to [7], [7] \to [5], [7] \to [6]\}$ and downward pointing arrows indicating the identity relation for data writes $W_{L_0 \to D_2} = W_{L_1 \to D_1} = \{[1] \to [1], [2] \to [2], [3] \to [3], [4] \to [4], [5] \to [5], [6] \to [6], [7] \to [7]\}$.

Given the loop chain abstraction of the Jacobi solver code, generating the partially ordered set of iterations is a straightforward process. The result is shown in Figure 1.3. Any

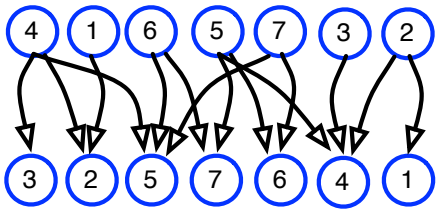


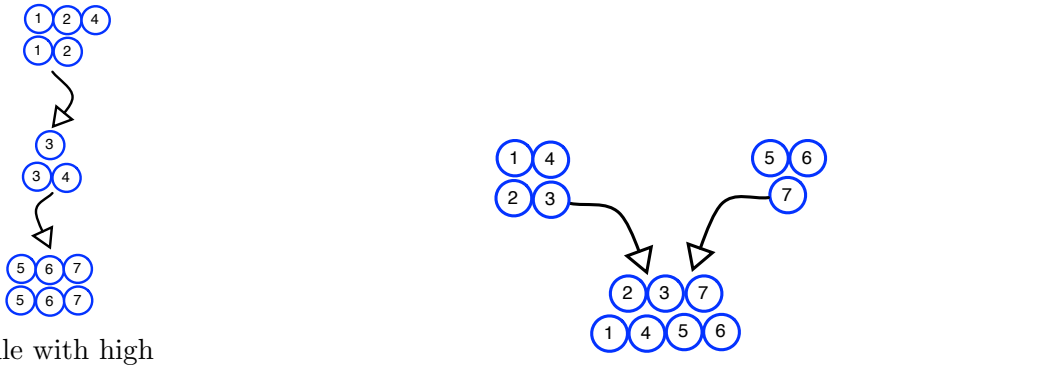
Figure 1.3: Partial ordering of iterations in the Jacobi loop chain

schedule for the loops in the loop chain that respects the partial ordering shown will be valid. This gives tremendous flexibility to the compiler, or in this case the runtime system to manipulate the iteration orderings to achieve a target mix of parallelism and data locality.

1.4 Introduction to Full Sparse Tiling

Given a loop chain and the implicit iteration partial ordering, the next step in generating a high performing schedule is selecting an iteration grouping and ordering that provides adequate parallelism while maximizing data locality. Full sparse tiling [55, 53] (FST) is a technique for accomplishing that goal. Full sparse tiling aims to cluster together iterations that access data in common, even across different loops in the loop chain. This converts eventual data reuse into more immediate data reuse, thereby improving temporal locality and in turn improving performance due to an improved cache hit rate. These iteration clusters are called *sparse tiles*. There is a partial ordering on when sparse tiles can execute relative to one another, which is simply an aggregation of the orderings imposed on individual iterations due to the data dependences captured by the data access relations. The partial ordering of sparse tiles can be expressed as a *task graph*.

Different full sparse tilings of loop chain iterations can vary the number of iteration clusters, called *sparse tiles*, used in the execution schedule. Schedules with fewer, and therefore larger, sparse tiles may have better locality than smaller tiles. Schedules with more tiles generally exhibit greater parallelism. Thus, by carefully constructing the tiles and controlling their size, full sparse tiling can balance parallelism and locality as appropriate for a given hardware system.



(a) A schedule with high locality and low parallelism.

(b) A schedule with more balanced locality and parallelism.

Figure 1.4: Different full sparse tilings yield different amounts of parallelism and locality.

Examples of some of the full sparse tilings possible for the loop chain presented in Figure 1.2 are given in Figure 1.4. The schedule shown in Figure 1.4a has excellent data reuse between iterations within a tile. However, only one tile can be executed at a time. The schedule shown in Figure 1.4b maintains some locality, but initially permits two tiles to execute in parallel. These are only two possibilities along the broad spectrum of possible full sparse tilings. To select between the different possible schedules, a human can manually evaluate the performance of different schedules and select the highest performing one. Alternatively, schedules can be selected based on predictions from a performance model or an autotuning approach can be used.

1.5 Problems with Single Purpose Approaches to Full Sparse Tiling

Previous to this work, each application of full sparse tiling was implemented for a specific type of code. For example, full sparse tilers were written for the MolDyn molecular dynamics simulator [53], a Jacobi solver such as was described in Section 1.3 [40], a Gauss-Seidel solver [54], and a matrix powers kernel [57, 25, 46]. Each of these tilers was developed largely from scratch and took considerable development effort. There was very little, if any, code reuse between the full sparse tilers. This led to a variety of code maintenance issues and unnecessary duplication of effort.

Because each tiler was purpose-built for a specific application, the tilers took advantage of properties of the given application to make assumptions about the tiling problem. For example, the Jacobi solver was written with the understanding that all vector writes would be done with an identity access pattern and that all read accesses would be specified by the non-zero pattern in the sparse matrix. These assumptions were valid and resulted in tiling code that, while inappropriate for other applications, was considerably simplified. However, in some cases, even a slight deviation from the original assumptions would result in invalid schedules being generated. For example, if a non-symmetric matrix were given to the Jacobi code, the tiler would ignore some storage related dependences and could sequence iterations illegally.

Similarly, assumptions were also made while writing the custom full sparse tiling code for the MolDyn molecular dynamics simulation. This simulation code consists of three loops in the loop chain. The first and last loops access memory through identity updates. It is only the center loop that has irregular accesses. Given this pattern, the custom code was able to partition the center loop and then assign iterations of the outer loops to tiles in a straightforward manner. This generates correct serial schedules. However, when these schedules were parallelized using the Jacobi task graph generation code directly, potentially illegal schedules were produced. This is because the Jacobi task sequencer does not consider reduction dependences within a loop, a situation that does not arise in the Jacobi solver but that is present in the center loop in MolDyn.

Finally, tile sizes resulting in optimal performance were determined for each application, and in some cases, for each input set, in an ad hoc fashion. These were usually determined through manual search, perhaps directed by a heuristic that made many simplifying assumptions. For example, some tile size selection code assumes that a seed partition's footprint should be equal to one-eighth of the mid level cache [54]. From this fixed starting point, the user would repeatedly run the application, making adjustments to the tile size until good performance was attained.

Experience with these application-specific full sparse tilers led to several conclusions. First, significant performance improvements, in some cases exceeding 50%, can be achieved by full sparse tiling [42]. However, it was also clear that writing a full sparse tiler is a laborious process that requires expert knowledge of tiling as well as domain knowledge. Second, little code could be reused between efforts due to deeply rooted assumptions that did not hold in other applications. Lastly, each of the tiler's parameters had to be tuned for the specific application in order to achieve sizable performance gains. No formal algorithm existed for finding the best performance for a given processor, core count, memory hierarchy, and scientific application. These problems together formed a formidable barrier to widespread adoption of full sparse tiling approaches.

1.6 Generalized Full Sparse Tiling

A major contribution of this research effort is the creation of a generally applicable full sparse tiling algorithm. As a precursor to designing such an algorithm, we identified several areas in which prior individual approaches were not general.

First, the efforts relied on pieces of the problem being described or specified in a particular way. For example, much of the Jacobi full sparse tiler depends on the data access pattern being described by the non-zeros within a compressed sparse row (CSR) formatted sparse matrix. Any problem in which dependences are not expressed as a sparse matrix immediately encounters incompatibilities.

Furthermore, a general full sparse tiler cannot assume a particular pattern or number of loops, nor can it rely on a specific type of data dependence between loops. It must be able to take all types of dependences into consideration, including storage related anti-, input, and output dependences.

In addition, if tiles are to be executed in parallel, dependences between tiles must also be understood. This therefore requires that the partial ordering between tiles, typically represented as a task graph, must honor an aggregation of the individual iteration dependences. In addition, the tiles must also be ordered to respect reduction dependences, even though

these dependences do not contribute to the partial ordering between loop iterations in a serial schedule.

Our *general* full sparse tiling algorithm was developed to squarely addresses these issues. A central feature of the algorithm is that it tiles code that has been expressed as a loop chain. No additional requirements are imposed as to the structure of the code and no additional information is needed. By using loop chains as an intermediary abstraction between the original code and the full sparse tiling algorithm, many of the pitfalls to generality that ensnared previous full sparse tiling efforts are neatly avoided.

Given a loop chain, the general algorithm is able to respect all types of data dependences, including storage related dependences and reduction dependences that were not explicitly analyzed in previous work. These dependences are completely derived from the data access relations and relative placement of loops as described in the loop chain. The algorithm accounts for all dependences while creating a serial schedule as well as a parallel task graph schedule. A complete description of the general full sparse tiling algorithm is presented in Chapter 3.

1.7 Understanding the Impact of Tile Size and Parallelism Under Generalized Full Sparse Tiling

Selecting the proper balance of parallelism and locality while full sparse tiling is an extremely challenging problem. Both parallelism and locality are heavily influenced by the number of tiles used when full sparse tiling, precluding the independent optimization of these two factors. If the task graph has low parallelism, particularly if the parallelism is lower than the number of cores available, the execution will be restricted to a subset of the total cores and therefore will have limited scalability. Likewise, if the average tile memory footprint greatly exceeds the target size, usually related to the size of a hardware cache, performance improvements will not be fully realized. This is due to the tile working set being larger than what can be supported by the cache hierarchy. In this case, while full sparse tiling improves locality, it does not sufficiently shorten the reuse distance of data to be smaller than the

cache size. Therefore, some understanding of the interaction between, or relative priority or importance of, parallelism and locality is useful when optimizing performance.

This work studies what metrics should be used when discussing and optimizing the tile size for improved locality. It also examines how the amount of parallelism in a task graph should be expressed. In addition to determining to what properties task graph execution time is sensitive, we also develop guidelines for balancing between locality and parallelism when selecting the tile count for a generalized full sparse tiling of a loop chain.

1.8 Summary of Contributions

This work spans a variety of important contributions, most of which are presented in greater detail later in this dissertation. Here is a succinct summary of the more significant items.

1. The Loop Chain Abstraction
   (a) A formal definition of a new programming construct called a *loop chain*.
   (b) A presentation of possible mechanisms for specifying the loop chaining. abstraction using libraries, domain specific languages, and pragmas.
   (c) A complete C++ loop chaining library and application programming interface for specifying loop chains.
2. Generalization of the Full Sparse Tiling Algorithm
   (a) A formal definition of generalized full sparse tiling for any loop chain that encompasses reduction and storage related dependences as well as flow dependences.
   (b) A C++ library that applies full sparse tiling to loop chains specified using the loop chain API.
   (c) Additional supporting algorithms and code:
      i. A novel graph partitioning algorithm, ParCubed, that is fast and parallelized for shared memory.

ii. A novel hypergraph partitioning algorithm, HyperParCubed, that is fast and parallelized for shared memory.

iii. A task graph execution engine for shared memory that builds upon the following existing parallel models:

A. Intel's Concurrent Collections programming model,

B. OpenMP parallel for loops,

C. OpenMP 3.0 tasks,

D. Cilk Plus spawn/sync language extensions,

E. pThreads library,

F. Intel's Threading Building Blocks flow graphs

3. A study of the forces affecting performance of a generalized full sparse tiled Jacobi solver including:

(a) The identification of irregular data footprint and median parallelism as important metrics for understanding performance in the context of generalized full sparse tiling.

(b) A study of the correlation between performance of generalized full sparse tiled code and cache size.

(c) Guidelines for selecting the amount of parallelism and the best irregular data footprint for optimal execution.

This dissertation work is a comprehensive effort to remove barriers to the wider adoption of full sparse tiling. Unlike previous full sparse tiling implementations, thanks to its use of loop chains, this work is useable by scientific domain experts and other programmers who are not specifically trained in parallel program optimization. A sparse tiling is automatically generated and relative tile dependences are captured in a task graph. This resulting graph is then executed in shared memory using the task graph executors specifically developed as part of this work. Thus, this dissertation is a complete sparse tiling package, from facilitating

a programmer's specification of a loop chain to generation of an optimized parallel schedule to efficient execution of the loops.

Chapter 2

Loop Chain Programming Abstraction

Each previous full sparse tiling effort focused on a single application. However, each of these applications had specific properties that made it a good candidate for full sparse tiling. Specifically, each had a series of loops that shared data. Each full sparse tiling research effort developed code and algorithms for analyzing the producer-consumer data dependency chains in that application's sequence of loops. By examining the similarities in these different instances of full sparse tiling, it became apparent that the dependency analysis code could be made general if application specific information describing the loops were expressed in a more abstract and universal fashion. In addition, if this abstraction were carefully designed, it could also be used by a much wider variety of reordering transformations than just full sparse tiling.

In Section 2.1, we further describe our motivation for developing the loop chain abstraction. We then define the abstraction and describe ways for specifying loop chains in Sections 2.2 and 2.3. Optimizations facilitated by loop chains, including, but not limited to, full sparse tiling, are discussed in Section 2.4. Sections 2.5 through 2.7 cover examples of loop chains that naturally occur in scientific codes, a comparison between loop chains and other related approaches, and a review of the limitations of the loop chain abstraction.

2.1 Motivation for the Loop Chain Abstraction

Typically, the bulk of the execution time of a scientific application is spent in loops. Thus, optimization efforts understandably focus on decreasing the execution time of these loops. Many of these optimizations consist of scheduling the iterations of the loops to achieve some objective. For example, iterations may be scheduled to improve data locality, minimize communication, or improve parallelism.

Reordering transformations are a class of optimizations that reschedule the execution order of iterations of a loop. In doing so, these transformations must respect the data dependences present in the original code. To do so, the dependences must be determined in some fashion. Approaches that rely on automatic code analysis often encounter challenges such as interprocedural data and pointer alias analysis that forces the use of conservative approximations. These issues can be avoided by requiring the programmer to explicitly enumerate the formal flow, anti, input and output dependences. However, such analysis is tedious, error prone, and is beyond the abilities of many programmers.

In this work, we have developed a new programming abstraction, *loop chains*, to address the problem of finding data dependences. Within the framework of a loop chain, a programmer specifies what data is accessed by each iteration of a loop through a *data access relation*. A data access relation is a simple relation from an iteration of a loop to the data elements that are accessed during the execution of that loop iteration. In many cases, this relation can be determined through simple code inspection. For example, for the code: `A[i] = k*B[C[i]]`, the relation would be from iteration [i] to element [i] of array A and from iteration [i] to element C[i] of array B. Listing data accesses is considerably easier for a programmer to do than to find the explicit data dependences. Having all the data accesses specified circumvents many of the issues that arise during interprocedural or alias analysis. The compiler, runtime, inspector, and/or other optimizer is then able to analyze these data access relations and find a partial ordering on the loop iterations based on the data dependences present in the original code. From these partial orderings, many different legal schedules can be derived. Some schedules can emphasize improved locality, while others provide a large degree of parallelism. Each of these optimized schedules results in a different grouping of the iterations.

To summarize, loop chains provide an incremental interface between programmers and compilers/run-time systems. They formalize and abstract a pattern that already exists in many scientific computing applications and domain-specific libraries. With the information

they provide, compilers and/or run-time systems are able to better balance between data locality and parallelism.

2.2 Formal Definition of the Loop Chain Abstraction

The loop chain abstraction consists of an ordered sequence of contiguous loops. Each loop must execute a fixed and known set of iterations. This excludes while loops or loops containing `break` statements. Additionally, there must not be any code between the loops in a loop chain. Loops in a loop chain must be doall parallel or reductions. There must not be any required ordering of iterations within a loop, such as would result from loop carried data dependences. Finally, there can be no conditional execution of any of the loops in a chain. Loops cannot be guarded with `if` statements or other statements that alter the control flow of the code. Any sequence of loops that meets these requirements can be expressed as a loop chain.

A loop chain abstracts away much of the detail of the original series of loops. What remains can be described using three components: iteration spaces, data spaces, and data access relations. Iteration spaces concisely describe all the iterations that a compiler or other optimizer must schedule. Similarly, the data spaces describe arrays or other chunks of memory accessed by code within the loop chain. Iterations and data space elements together are the building blocks that are grouped, reordered, or distributed to effect a specific optimization. The data access relations define the relationship between iterations and data. These relations expose not only limitations on iteration reordering due to data dependences but also reveal opportunities to improve locality. The notation used to refer to these components is given in Table 2.1 and each is described in the following sections.

2.2.1 Iteration Spaces

Each loop has a well defined *iteration space*, $L_0, \ldots, L_{N-1}$, where N is the number of loops in the chain. In the context of loop chains, the term iteration space refers to the unordered set of executed iterations of a loop body, with each iteration identified by an

Table 2.1: A summary of the symbols used to represent loop chains.

Symbol	Description
L	A sequence of loops $(L_0, L_1, ...L_l, ...L_{N-1})$.
l	An identifier for the l^{th} loop in the sequence.
L_l	The iteration space associated with the l^{th} loop in the sequence.
$\vec{i}$	A vector that identifies a specific iteration within a loop.
D	A set of data spaces $\{D_0, D_1, ...D_d, ...D_{M-1}\}$.
D_d	A specific data space. For example, D_0 may correspond to a data array X.
$\vec{d}$	A vector that identifies a specific element within a data space.
$R_{L_l \to D_d}(\vec{i})$	A relation between iterations and the data they read.
$W_{L_l \to D_d}(\vec{i})$	A relation between iterations and the data they write.

iteration vector or tuple containing the values of all index variables for that iteration. For example, in the case of a simple for loop: `for (i=0; i < 5; ++i) {...}`, L_0 would refer to the five iterations of the loop, $L_0 = \{[0], [1], [2], [3], [4]\}$.

One could define a loop's iteration domain as a contiguous range, a polyhedral set, or an explicitly enumerated set of items. For example, if iterating over the edges of an unstructured mesh, the domain may be defined as $\{[edge_0], [edge_1], \ldots, [edge_{numEdges-1}]\}$. Likewise, a container can implicitly define an iteration space as a traversal over all elements in the container. Threading Building Blocks [35], as well as the OP2 unstructured mesh library [11] and the Chombo library [21], both discussed in Section 2.3, all support this type of iteration. The loop chain abstraction does not require that a particular definition of the domain of iterations be used, just that the iteration space domain is well defined.

Observe that an iteration space's set of iterations is unordered. This implies that the original loop must be either doall parallel or else a reduction. There can be no loop carried dependences that impose a relative order on the execution of iterations within a loop. Further note that in the case of iteration spaces, the subscript indicates ordering within the chain, e.g. L_0 precedes L_1 and so forth. This means that any data dependence is between iterations of different loops and that the original ordering of those two loops is known. Taken together, these two requirements greatly simplify dependence analysis. For example, a flow dependence is always from a write in one loop to a read in a higher numbered loop.

2.2.2 Data Spaces

Each data structure accessed within the loop chain must be formally declared as a *data space*, denoted $D_0, D_1, \ldots, D_m$. For data spaces, the subscript numbering does not denote an ordering or placement in a sequence and is only used to uniquely identify the various spaces. Each data space must have a well defined domain. Typical data spaces are one dimensional or multidimensional arrays, though associative arrays, maps, or sets are also possible. Analogous to iteration spaces, a data space refers to a data structure and the complete set of valid indices in the structure's domain or index space. For example, a data space, D_0, for the ten element array A, would be $D_0 = \{A[0], A[1], \ldots, A[9]\}$. While all one dimensional, ten element arrays share the same domain, each would have a unique data space. A data space on a map, `paint`, indexed by color names might be $D_1 = \{\texttt{paint}(red), \texttt{paint}(green), \ldots, \texttt{paint}(black)\}$. Note that the values stored in the referenced data structure are irrelevant and are not part of the data space.

2.2.3 Data Access Relations

As part of a loop chain definition, each loop in the loop chain must declare the data items that are read or written by each iteration of the loop. This is accomplished through the use of *data access relations* (DARs). Data access relations are mathematical relations between iterations in an iteration space, L_l, and elements in a data space, D_d. Read relations

Table 2.2: Enumeration of the data access relations for the example given in Figure 1.2.

DAR on Loop 1	Elements	DAR on Loop 2	Elements
$R_{L_1 \to D_{Uodd}}(1)$	{2}	$R_{L_2 \to D_{Ueven}}(1)$	{2}
$W_{L_1 \to D_{Ueven}}(1)$	{1}	$W_{L_2 \to D_{Uodd}}(1)$	{1}
$R_{L_1 \to D_{Uodd}}(2)$	{1}	$R_{L_2 \to D_{Ueven}}(2)$	{1}
$W_{L_1 \to D_{Ueven}}(2)$	{2}	$W_{L_2 \to D_{Uodd}}(2)$	{2}
$R_{L_1 \to D_{Uodd}}(3)$	{4}	$R_{L_2 \to D_{Ueven}}(3)$	{4}
$W_{L_1 \to D_{Ueven}}(3)$	{3}	$W_{L_2 \to D_{Uodd}}(3)$	{3}
$R_{L_1 \to D_{Uodd}}(4)$	{2,3}	$R_{L_2 \to D_{Ueven}}(4)$	{2,3}
$W_{L_1 \to D_{Ueven}}(4)$	{4}	$W_{L_2 \to D_{Uodd}}(4)$	{4}
$R_{L_1 \to D_{Uodd}}(5)$	{4,6}	$R_{L_2 \to D_{Ueven}}(5)$	{4,6}
$W_{L_1 \to D_{Ueven}}(5)$	{5}	$W_{L_2 \to D_{Uodd}}(5)$	{5}
$R_{L_1 \to D_{Uodd}}(6)$	{7}	$R_{L_2 \to D_{Ueven}}(6)$	{7}
$W_{L_1 \to D_{Ueven}}(6)$	{6}	$W_{L_2 \to D_{Uodd}}(6)$	{6}
$R_{L_1 \to D_{Uodd}}(7)$	{5,6}	$R_{L_2 \to D_{Ueven}}(7)$	{5,6}
$W_{L_1 \to D_{Ueven}}(7)$	{7}	$W_{L_2 \to D_{Uodd}}(7)$	{7}

are described by $R_{L_l \to D_d}(\vec{i})$ and write relations by $W_{L_l \to D_d}(\vec{i})$. The access relations indicate which data locations in data space D_d that an iteration $\vec{i} \in L_l$ accesses.

The DARs can be uninterpreted at compile-time, but must be known at runtime before the execution of the loop chain begins. This may be the case when the data access relation is dependent on input data, such as an unstructured mesh's topology or the placement of non-zeroes in a sparse matrix. Table 2.2 shows the data access relations for the example Jacobi solver and specific sparse matrix used in Figure 1.2

The data access relations are the critical piece of the loop chain abstraction. The DARs fully specify the data dependences between iterations of loops in the loop chain. The DARs can be used directly to determine if two iterations access the same data and therefore potentially have a data dependence or the DARS can be transformed into formal data dependency relations.

When specifying the access relations for a loop body, the programmer must include any data accesses that result in data dependences between iterations. This includes accesses that may be made from within procedures called from the loop body. Because the access relations are specified by the user, the granularity of the access relations can be user-determined. For

example, in a molecular dynamics application, the access relations may be from a loop iteration to a single directional component of the velocity vector of an atom, to any element of the velocity vector of an atom, or simply to the entire data structure for a particular atom, including velocity, force, position, etc. The programmer can also decide if a data access present in the code is relevant to the loop chain. For example, if a loop accesses data that remains invariant throughout the lifetime of the loop chain, the user may or may not choose to include those accesses. Some optimizations may benefit from that information, while others may not be impacted by the absence of that information.

2.3 Methods for Specifying Loop Chains

While it may be possible for a loop chain abstraction to be automatically extracted from source code by a compiler or other tool, in most cases a programmer will be required to specify all or part of the loop chain information. There are a variety of mechanisms that can be used to accomplish this task. These approaches range from an explicit description of the loop chain using an application programming interface or set of pragmas, to more implicit methods using domain specific languages or language extensions.

2.3.1 Application Programming Interfaces

As part of this dissertation, we wrote a complete application programming interface and C++ library called GROUT for specifying loop chains. The objects in the class library map directly to the elements in a loop chain definition. Figure 2.1 shows an example of using this API on the Jacobi solver code presented earlier in Figure 1.1. Line 2 creates a loop chain object. Line 6 instantiates an iteration space with a continuous range from 0 to `numberOfRows-1` as its domain.

Lines 9 and 10 declare the data spaces for the UOdd and UEven arrays. The library supports several types of data access relations. The most general explicitly enumerates the data items associated with each iteration. Other access relations are provided for common, but more specialized, relations. The read data access relation declared on line 13 is of type

```
1  // declare a loop chain
2  LoopChain chain0;
3
4  // create the iteration space
5  int numberOfRows = matrix->getNumRows();
6  ContiguousIterSpace iterAllRows(0,numberOfRows-1);
7
8  // create the data spaces
9  DataSpace UOdd(numberOfRows);
10 DataSpace UEven(numberOfRows);
11
12 // Create the access relations
13 CSRAccessRelation relReadUOdd(iterAllRows, UOdd, matrix,
14                               AccessType::READ);
15 IdentityAccessRelation relWriteUPrime(iterAllRows, UEven,
16                               AccessType::WRITE);
17
18 // Assemble the loop
19 Loop loopUpdateUEven(bodyJacobiUpdateUEven, iterAllRows);
20 loopUpdateUEven.addAccessRelation(&relReadUOdd);
21 loopUpdateUEven.addAccessRelation(&relWriteUEven);
22
23 // add the loop to the loop chain
24 chain0.addLoop(loopUpdateUEven);
```

Figure 2.1: The kernel of a sparse Jacobi solver loop chained using the LoopChain API.

`CSRAccessRelation` and is an example of a specialized access relation that is derived from the non-zero pattern in the provided sparse matrix. The write relation on line 15 is a simple identity relation. These data access relations are attached to the loop `loopUpdateUEven`, which is then added to the loop chain on line 24.

Using the LoopChain API requires knowledge of loop chains and their definition. It requires 16 lines to fully specify the chain of two loops found in the Jacobi solver kernel. This approach to specifying loop chains is best used by experts or as the target of a tool, such as a source-to-source translator.

2.3.2 Pragmas

Pragmas can be used to annotate a program's source code, thereby delimiting the loop chain and providing information about data and iteration spaces and access relations. Figure 2.2 shows code taken from the Jacobi sparse matrix solver presented earlier in Figure 1.1. On lines 1 through 3, a hypothetical `chain` pragma, similar to OpenMP's pragmas, is shown. It declares the loop as part of a loop chain named `chain0`. For this example, the proposed system would infer the iteration space from the loop bounds.

```
1  #pragma chain chain0\\
2  access( read, UEven, function(JA[p], IA[i] <= p < IA[i+1])) \\
3  access(write, UOdd, identity(i))
4  for (int i=0; i<numrows; i++) {
5    double diag = 1.0;
6    UEven[i] = 0.0;
7    for (int p=IA[i]; p < IA[i+1]; p++) {
8      int j = JA[p];
9      if (j==i) { diag = A[p]; }
10     else { UEven[i] += A[p] * UOdd[j]; }
11   }
12   UEven[i] = (F[i] - UEven[i]) / diag;
13 }
14   ...
```

Figure 2.2: The kernel of a sparse Jacobi solver loop chained using hypothetical OpenMP style pragmas.

Lines 2 and 3 show how data access relations could be specified. Line 2 is a read relation on the `U` array. The relation itself might be a function declared using a `function` keyword describing the access as `JA[p]`, where the value of `p` is given by the inequality `IA[i] <= p < IA[i+1]`. The identity write relation on the `Uprime` array is given on line 3. The second loop in the loop chain would be specified with a similar pragma.

Using a pragma-based approach may require only two pragmas to specify the loop chain in the Jacobi solver. However, the programmer must still be aware of the specifics of how loop chains are defined and the syntax for the pragmas might be considered cumbersome. Using pragmas also requires a compiler that can recognize and process the pragma appropriately. More research is necessary to determine the best pragma syntax that is both useable for programmers and feasible within existing compiler frameworks.

2.3.3 Domain Specific Languages

If loop chains are used in conjunction with an existing domain specific language (DSL), the task of specifying the needed loop information can be considerably simplified. Consider the OP2 (Oxford Parallel Library, version 2) [11] DSL. OP2 is designed for parallel computation on unstructured meshes. Its main feature is a parallel loop structure, `op_par_loop`. This parallel loop call takes as arguments an iteration space as well as some number of maps that go between iterations within the provided iteration space and elements in data struc-

```
1  program meshProgram
2    call op_decl_set (vertices, meshFile, "vertices")
3    call op_decl_set (cells, meshFile, "cells")
4    call op_decl_set (edges, meshFile, "edges")
5
6    call op_decl_dat (vertices, 6, vertexData, meshFile, "vertexData")
7    call op_decl_dat (edges, 6, edgeData, meshFile, "edgeData")
8
9    call op_decl_map (edges, vertices, 2, edges2Vertices, meshFile, "edges2Vertices")
10   call op_decl_map (cells, vertices, 2, cells2Vertices, meshFile, "cells2Vertices")
11
12 begin_loopchain ()
13
14   ! loop over edges
15   call op_par_loop (edges, kernel1, &
16    op_arg_dat (temp, 1, edges2vertice, OP_INC), &
17    op_arg_dat (temp, 2, edges2vertice, OP_INC), &
18    op_arg_dat (x, -1, OP_ID, OP_READ))
19
20   !loop over cells
21   call op_par_loop (cells, kernel2, &
22    op_arg_dat (temp, 1, cells2vertices, OP_READ), &
23    op_arg_dat (temp, 2, cells2vertices, OP_READ), &
24    op_arg_dat (temp, 3, cells2vertices, OP_READ), &
25    op_arg_dat (res, -1, OP_ID, OP_WRITE))
26
27 end_loopchain ()
```

Figure 2.3: A kernel loop chained using OP2 loop constructs.

tures. The parallel loop mechanism works by calling the specified subroutine and passing it a single element of the iteration space. Additional arguments are retrieved from data sets using the iteration element and the data maps and passed to the subroutine.

Consider the example shown in Figure 2.3. The code shown here is two `op2_par_loop` calls that have been loop chained together. Lines 2 through 4 declare sets over which iteration can occur. These are the vertices, cells, and edges of the underlying mesh. Lines 6 and 7 identify data that is associated with vertices and edges, respectively. Lines 9 and 10 define mappings from edges to vertices or from cells to vertices.

If these OP2 DSL calls are considered from the vantage point of loop chains, analogies can be drawn. The `op_decl_set` calls of lines 2-4 are iteration space definitions. The `op_decl_dat` calls (lines 6-7) identify data spaces and the `op_decl_map` calls (lines 9-10) define data access relations. However, note that all these lines are present in OP2 code and are not specific to loop chaining. The only lines added for loop chains are lines 12 and 27, which declare the start and end of the loop chain.

Other DSLs, such as Chombo Fortran [21] for thermodynamics computations on structure meshes, also provide a similar level of ease of use for loop chaining.

2.4 Optimizations Enabled by Loop Chains

The loop chain's data access relations describe the data accesses that occur within each iteration of each loop in the chain. Based on that information, a compiler, runtime execution engine, or inspector can determine how best to schedule and place iteration execution as well as determine data layout and distribution.

For example, given a loop chain, data within sequential data spaces like arrays can be reordered to place data items such that they are stored in the order they are accessed. One algorithm for accomplishing this type of data reordering is the consecutive packing (CPACK) algorithm [28], which packs together data based on the first time it is accessed. Similarly, if iterations are grouped together into tiles or blocks, data can be reordered and packed such that all data items accessed by iterations within a tile are located in adjacent memory locations. These are just two examples of data reordering transformations that can be implemented using only data access relations and iteration orderings.

The loop chain abstraction also enables iteration reordering transformations, including those that mix iterations from different loops. One simple example is loop fusion [2], an optimization in which the *nth* iteration of two or more loops are either combined into a single loop iteration or are scheduled one after another. A simple examination of the iteration partial ordering can determine if this optimization is valid and a review of the data access relations can show if loop fusion would result in improved data locality.

Full sparse tiling, introduced in Section 1.4, also reorders iterations across loops in an effort to improve locality. The seed iteration space is partitioned using a graph or hypergraph derived from the data access relations. These seed partitions are then grown to include iterations of other loops, once again under the direction of the data access relations. Figure 1.4 shows two schedules generated by full sparse tiling for the Jacobi solver example given in Figure 1.1.

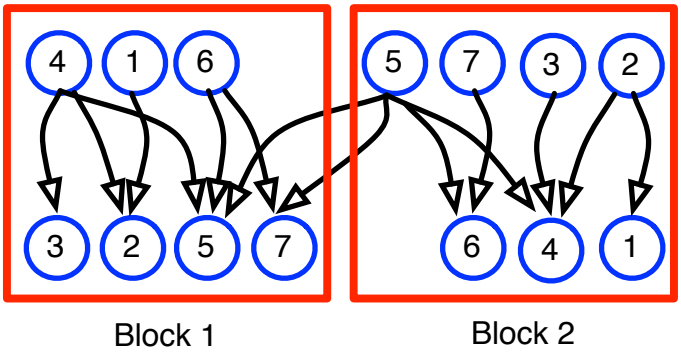


Figure 2.4: A distributed memory work distribution across two nodes. Only element 5 of the Ueven vector, written by iteration 5 of the first loop, needs to be communicated from node 2 to node 1. All other data can be placed during initial data distribution.

A wide range of parallelization techniques can also be applied to loop chains. Different schedules or iteration groupings can be generated, depending on the type of parallel hardware being targeted. For example, a shared memory tiling can focus on memory locality and grouping the iterations based on in-core cache size.

Unlike shared memory parallelization, a distributed memory parallelization must consider internodal communication and data distribution. Fortunately, using data access relations, the data needed by each iteration assigned to a distributed memory node or address space is immediately known. Any data a node needs that are invariant over the life of the loop chain can be initially distributed using the appropriate communication. Since it is clear what data is altered during execution of the loop chain and by what node it is written, a communication schedule can be generated directly. This data being sent between nodes can then be aggregated into a small number of communication requests.

For example, in Figure 2.4, a distributed memory work distribution is shown for the Jacobi example of Figure 1.1. This schedule is easily derived from the partially ordered iterations of Figure 1.3. All communication between the two nodes is indicated by arrows crossing between nodes. Only data item `UEven[4]` needs to be sent between the nodes. Work by Basumallik et al. [7] takes an approach similar to this to aggregate and overlap communication and computation in a distributed memory system.

Parallelization approaches that involve work offload to a general purpose graphics processing unit (GPGPU) require much the same data distribution as traditional distributed memory parallelization. These approaches, however, may have additional requirements in terms of communication and synchronization between threads on the GPGPU. Because loop chains require that all iterations of a particular loop have no imposed ordering, no intraloop synchronization is required except in the case of reductions. Between loops, synchronization is only needed as determined by the partial ordering. This allows the computation to be broken into thread blocks that communicate via shared memory and only need to communicate via global memory rarely. Consider again Figure 2.4. The iterations assigned to node one could be executed within a thread block and those assigned to node two could be executed in another thread block. Each iteration in a given loop could be assigned to a different thread within the block. Between loops, the GPGPU would need to synchronize using hardware synchronization, such as is requested via Nvidia's CUDA language primative `__syncthreads()`. The only communication needing global memory is the passing of `UEven[4]`. Synchronization between thread blocks requires scheduling assistance from the CPU to ensure that the thread block handling block two is not launched on the GPGPU until after block one has completed.

The examples given in this section are representative of the wide range of optimizations and code transformations that are enabled by loop chains. Shared memory, distributed memory, and GPGPU parallelization methods are all facilitated by using the loop chain abstraction. Data and iteration reordering transformations also can be expressed within the context of loop chains.

2.5 Examples of Loop Chains Present in Existing Scientific Codes

Commonly, data in scientific applications is organized as structured or unstructured meshes, grids, or geometric subdivisions, such as boxes, of a larger three dimensional space.

These subdivisions may come from finite element or finite volume approaches to solving partial differential equations.

In order to promote sustainability and reusability of code, scientific applications are often architected as a series of passes over some data structures. Each pass applies a particular function or computes one type of value, e.g. temperature, pressure, velocity, position, and so forth. Data from one pass often feeds the next pass through a producer-consumer relationship. Multiple passes may also be the result of applying a function to each of multiple dimensions, such as the force in the x, y, and z dimensions. Rather than combine passes, the passes are kept distinct so that the code can be reused and so that the code does not become intertangled [37, 21].

One example of this code structure is given in Figure 2.5, which shows the central kernel of a molecular dynamics benchmark. Here a sequence of three loops is found inside an outer time step loop.

Figure 2.6 shows another example of a loop chain. This example is drawn from the *airfoil* computational fluid dynamics test program [10, 11]. A series of three loops in enclosed in an outer time step loop. The inner loops compute the change in area of a mesh cell, the flux residual for each edge, and the flow through each cell, respectively.

Even though this example is drawn from an entirely different scientific domain and is solving a distinctly different problem, the overarching design of the code very closely resembles the molecular dynamics simulation given in Figure 2.5. Other similar occurrences of series of loops can be found in thermodynamics simulations [21].

In some cases, a series of loops is the result of unrolling a loop by some factor. What was previously one loop becomes a series of related loops after unrolling. This was seen in the case of the ping-pong version of the Jacobi solver seen in Figure 1.1. In this code, the main convergence loop has been unrolled by a factor of two, resulting in a series of two sequential loops. These unrolled loops form a loop chain. Greater unrolling factors generate progressively longer loop chains.

```
1  for (int timestep=0; timestep < maxTimesteps; ++timestep)
2  {
3     // do the per-atom position updates
4     for (int i=0; i < numAtoms; i++)
5     {
6       x(i) = x(i) + vhx(i) + fx(i);
7       y(i) = y(i) + vhy(i) + fy(i);
8       z(i) = z(i) + vhz(i) + fz(i);
9
10      // clear the force vector
11      fx(i)  = 0.0;
12      fy(i)  = 0.0;
13      fz(i)  = 0.0;
14    }
15
16    // do the per-interaction force updates
17    for (int ii=0; ii < ninter; ii++)
18    {
19      int i = inter1(ii);
20      int j = inter2(ii);
21
22      // compute the force between interacting atoms
23      forcex = f(x(i), x(j))
24      forcey = f(y(i), y(j))
25      forcez = f(z(i), z(j))
26
27      fx(i)  += forcex ;
28      fy(i)  += forcey ;
29      fz(i)  += forcez ;
30
31      fx(j)  -= forcex ;
32      fy(j)  -= forcey ;
33      fz(j)  -= forcez ;
34    }
35
36    // do the per-atom velocity updates
37    for (int i=0; i < numAtoms; i++)
38    {
39       ...
40      vhx(i) += fx(i);
41      vhy(i) += fy(i);
42      vhz(i) += fz(i);
43    }
44
45 }
```

Figure 2.5: The kernel of the MolDyn molecular dynamics benchmark. Within the outer time step loop is a sequence of three inner loops. They compute the position, interactive force, and velocity of atoms in the simulation, respectively, and form a classic loop chain.

```
1
2  int edgesToCells[numEdges][2]; // maps from edge to cells on either side
3
4  for (int timestep=0; timestep < maxTimesteps; ++timestep)
5  {
6
7     // for each cell, compute the change in area based on flow q
8     for (int cell=0; cell < numCells; ++cell)
9     {
10       updateArea(q[cell], area_dt[cell]);
11    }
12
13    // for each edge, update the flux residual for cells on either side of edge
14    for (int edge=0; edge < numEdges; ++edge)
15    {
16       calcFluxResidual(q[edges2Cells[0]], q[edges2Cells[1]],
17                        area_dt[edges2Cells[0]], area_dt[edges2Cells[1]],
18                        residual[edges2Cells[0]], residual[edges2Cells[1]]);
19    }
20
21    // update the flow field per cell based on the area and residual
22    for (int cell=0; cell < numCells; ++cell)
23    {
24       updateFlowField(area_dt[cell], residual[cell], q[cell]);
25    }
26
27 }
```

Figure 2.6: The kernel of the *airfoil* computational fluid dynamics benchmark. Within the outer time step loop is a sequence of three inner loops. They compute the change in area, flux residual, and flow field, respectively, and form a classic loop chain.

2.6 Prior Work Related to Loop Chains

The loop chaining abstraction complements the application of previously developed automated code transformation strategies. The abstraction promises to enable the same level of performance as manual approaches, while remaining portable to new architectures and not requiring users to define explicit tasks. This section describes work previously done to expose asynchronous parallelism by scheduling manually defined tasks, scheduling automatically discovered tasks, and by rewriting existing algorithms to avoid communication.

2.6.1 Programming Models Using User Defined Tasks

The following describes several projects that aim to expose the available parallelism in code through the application of new programming models that involve the programmer defining tasks.

The Tarragon system [20] provides an API for programmers to create a task graph with edges labeled with what data needs to be communicated from one task to the next. Distributed memory parallelism is handled by having any data on an edge be sent as a message. The run-time system schedules the task graph so that there is computation and communication overlap. This approach requires the programmer to decide what computations will be aggregated into tasks.

In the SuperMatrix work [16], the programming model is to specify matrix computations as submatrix computations. The run-time system can determine how many times subdivision occurs and then can create a task graph at run-time to expose asynchronous parallelism. This solution removes the burden of selecting task sizes from the user, but is more restricted than the loop chaining programming abstraction we are proposing that will work more generally for sequences of data parallel loops.

Concurrent Collections (CnC) is a programming model based on data-flow programming models. The programmer expresses the computation in terms of high-level application specific components, partially-ordered only by data and control flow constraints. Since data items are restricted to a single assignment model, the run-time system has more flexibility in the scheduling of steps. This is a "managed computation" approach where data reuse and scheduling need to be managed by the run-time system. To achieve performance, aggregation of computations into groups that access similar data is still necessary [17]. The loop chaining abstraction better exposes the partially ordered set of fine-grained computations so that this aggregation into tasks can be done automatically. Additionally, loop chaining is a more iterative approach for converting data parallel programs into programs with asynchronous computation.

StarPU [3] is a runtime system providing a high-level, unified execution model tightly coupled with a data management and communication library. The main goals of StarPU are to provide scientific kernel designers with a convenient way to generate parallel tasks over heterogeneous hardware and to easily develop and tune scheduling algorithms. The StarPU work focuses on scheduling to both CPUs and GPUs and dynamically selecting the most

efficient hardware type for a given task. In this scheme, tasks, called *codelets*, are manually defined for different architectures. The loop chaining abstraction is a more incremental approach for defining tasks and could dovetail with the StarPU work nicely.

2.6.2 Automatic Approaches for Task Detection

An alternative to providing a new programming model is to automatically examine application code and create task graphs.

Iteration space slicing [48, 49] is an optimization that would be applicable in loop chains where the access relations are affine. Overlapping iteration space slices would have duplicate computation, but a large amount of parallelism.

A method of partitioning loops with irregular accesses for parallel execution on distributed systems was presented in [7]. This method takes advantage of OpenMP directives and combines both compile-time and runtime techniques to translate OpenMP code to MPI. Data accesses and computations are reordered in order to increase the overlap of communication and computation. However, loops are examined individually and locality between different loops is not examined. This work could leverage the loop chaining abstraction to reduce the program analysis necessary.

The code generation technique presented in [50] examines the data affinity among loops and partitions the execution with the goal of minimizing communication between processes, while maintaining load balancing. This technique is also aimed at irregular applications executing in a distributed memory environment, and it could leverage the loop chaining abstraction.

2.6.3 Communication Avoidance

Several techniques for scheduling across loops to avoid communication have been prototyped by hand or with specialized code generators.

In structured codes, using multiple layers of halo, or ghost, cells is a common optimization [6], but when done by hand breaks the typical software modularity in large applications.

Structured codes can also use overlapped tiling techniques that reduce communication at the expense of performing redundant computation [59]. These techniques work by dividing a loop nest's iteration space into a number of overlapping regions called tiles. These tiles are shaped such that each of them can execute in parallel without requiring communication. Several researchers have examined using overlapped communication techniques within single loop nest computations [44, 43, 19]. We believe that an overlapped tiling optimizer could use information from the loop chaining abstraction to do overlapped tiling across loops.

For unstructured codes, there has been various inspector/executor strategies [45] that reschedule across loops to improve data locality while still providing parallelism [29, 54, 25, 40]. The term communication avoidance was coined by Demmel et al. [25] to refer to such schedules, some of which have processors perform some amount of overlapped computation to avoid communication. All of these techniques that schedule across loops potentially could be more easily automated if the loop chaining abstraction were used.

2.7 Limitations of Loop Chains

While loop chains are a capable tool for describing a specific class of loop sequences, the nature of loop chains imposes some limitations on the types of code loop chains can represent.

For example, the requirement that all loops in a loop chain be doall parallel or reductions precludes the chaining of any loop containing loop carried dependences or having any ordering of iterations within a single loop. This prevents the chaining of the key loops in kernels such as the sparse Gauss-Seidel solver, which has a loop carried dependence. Sparse Gauss-Seidel, can, however, be scheduled using a custom full sparse tiler [55]. This indicates that some sequences of loops that share data and that are optimizable using techniques similar to those presented in this dissertation cannot be expressed using the current loop chain definition.

To handle these types of intra-loop dependence patterns, the loop chain definition would need to be extended to include a partial ordering on iterations within a loop. This is a straightforward addition to the loop chain abstraction, but does have some disadvantages.

First, the programmer or automated system would have to detect and understand these dependences. A key principle in the design of the loop chain abstraction was to avoid having to deal with dependence analysis, making this addition an undesirable departure from that goal. Second, any optimization that processes loop chains would have to support the significant added complexity of intra-loop dependences. For these reasons, intra-loop dependences are not presently supported.

Another limitation of loop chains is the requirement that there be no code between the loops in the chain; the loops must be contiguous. This poses a challenge to the use of loop chains in some scientific applications. For example, there may be code to do checkpointing or in-situ visualization placed between loops in an otherwise chainable series of loops. These codes may involve strictly ordered writing of data to a network or storage device.

If enough loops are present in the chain, it may be possible to split the chain into two chains, one containing the loops before the intervening code and one containing the loops after the interrupting code. This may reduce the efficacy of some locality-improving optimizations, but allows some chaining to take place.

In some cases, code between loops can be coerced into the loop chain model. This can be accomplished by introducing a loop with a single iteration. This loop has a data access relation indicating that the single loop iteration reads all the data elements to which the preceding loop writes. These reads may be actually present in the singleton iteration or can be contrived. This will serialize the execution of the single loop iteration after the execution of all iterations of the preceding loop. A new data space with one element is then added to the loop chain and a write data access relation is added stating that the singleton iteration writes to this data item. A read data access relation is also added to the loop following the intervening code indicating that all iterations of the following loop read from this single item. In this way, code between loops can be forced to execute at the appropriate time. Note however that this process essentially creates a barrier between the loops and greatly reduces the parallelism of any schedule found for the loop chain.

An additional limitation on loop chains is their lack of explicit support for conditional execution. All iterations of all loops within a loop chain must execute each time the chain executes. However, this is largely a performance concern, rather than a limitation on expressivity, because each loop body is permitted to contain conditionals. If a loop body is entirely contained within an *if* statement, the same effect is achieved as if the loop itself were within a conditional. Note that if loop bodies contain conditional statements, data access relations must capture all possible memory accesses in any execution path through the loop body code. This may lead to overly conservative, but legal, execution orderings.

Overall, the loop chain abstraction is a careful tradeoff between expressivity and simplicity. If it were to be extended, the expressivity would increase, but it most likely would be more difficult for application programmers and optimization developers to use. If it were simpler to use, it would not have enough information to succeed as a gateway to a partial ordering on loop iterations. Future work may further refine the level of abstraction of a loop chain.

Chapter 3

Generalized Full Sparse Tiling

One class of optimizations that can take advantage of the loop chain abstraction are inspector/executor (I/E) schemes [52]. Under an I/E scheme, data is inspected at run time to determine indirect memory access patterns that are not fully determined at compile time. Based on these patterns, loop iterations are reordered. *Full sparse tiling* (FST) [55, 54, 40] is an I/E optimization that improves temporal and spatial locality by placing the execution of loop iterations that access the same data, even across different original loops, together into a scheduling entity called a *sparse tile*. By carefully constructing the sparse tiles and controlling their size, full sparse tiling can balance parallelism and locality as appropriate for a specific hardware system. Special purpose implementations of full sparse tiling have been written for particular codes and are described in Section 3.1, but these approaches made specific assumptions and could not be reused to sparse tile a different application. We discuss the challenges to reusing these codes in Section 3.2. To resolve these issues and greatly improve the accessibility of full sparse tiling, we developed the *generalized* full sparse tiling algorithm (gFST), which we present in Section 3.3. The validity of the algorithm is discussed in Section 3.4.

3.1 Prior Single Purpose Approaches to Full Sparse Tiling

Sparse tiling techniques were initially introduced by Douglas et al. [29] to parallelize computations over unstructured meshes. They referred to their approach as *unstructured cache blocking*. Unstructured cache blocking was applicable in a loop that iterated over an unstructured mesh. The dependence pattern in these iterative computations were nearest-neighbor, similar to those found in the Jacobi solver. The mesh was partitioned and that

partitioning was the tiling in the first iteration of the loop over the mesh. Tiles would then shrink by one layer of vertices for each iteration of the loop. This shrinking represented what parts of the mesh could be updated in later iterations of the loop without communicating with processors doing other tiles. The unstructured cache blocking sparse tiling needed a serial cleanup tile to complete all remaining work at the end of the computation.

Mark Adams [1] developed an algorithm that could be considered sparse tiling *within* a loop to parallelize sparse Gauss-Seidel computations. The inspector phase would partition the mesh and then create a partial ordering between partitions in the mesh. The partial ordering was represented with a task graph and parallelism could be found in the level sets of the task graph.

The term sparse tiling and an algorithm called *full sparse tiling* were introduced by Strout et al. [54, 56] in the context of the Gauss-Seidel algorithm and in [53] in the context of the moldyn benchmark. In [54, 56] the tiles extended across iterations of the outer loop as in the Douglas work [29], but the tiles fully covered the iteration space thus avoiding a large cleanup tile. This work originally focused on using sparse tiles to improve data locality, but later was extended for parallelism by creating a task graph where each tile was a task and parallelism was found in level sets as in the Adam's work [1]. In addition to Gauss-Seidel, these techniques were applied to a sparse Jacobi solver [41, 40, 39] and a molecular dynamics benchmark [53].

Researchers at Imperial College of London, in conjunction with Rolls-Royce, have recently begun implementing full sparse tiling within the Oxford Parallel Library Version 2 (OP2) [10, 11] framework. Their approach differs from other full sparse tiling work in that they initially partition data, rather than iterations. They use a graph coloring approach to find work that can execute in parallel and do not explicitly use a task graph. This work has led to the full sparse tiling of a computational fluid dynamics benchmark [38].

Other sparse tiling techniques have also been investigated under the umbrella term *communication avoiding* [25, 46]. The communication avoiding work describes an overlapping sparse tiling approach that covers the full iteration space and due to the overlap is able to

start execution of all the tiles in parallel. The tradeoff here is that some computation is done redundantly. In [46], the serial implicit technique is presented, which is equivalent to full sparse tiling, but is specifically for the matrix powers kernel.

In summary, there have been a number of sparse tiling techniques employed to improve the performance of specific applications. These approaches have been successful in delivering improved performance. This dissertation builds on the foundation laid by this prior work and generalizes it to any series of loops expressible using the loop chain abstraction.

3.2 Issues With Generalization of Full Sparse Tiling

In Section 3.1, in addition to other tiling techniques, we surveyed a number of existing full sparse tiling algorithms. Each of these efforts has successfully applied the basic techniques of full sparse tiling to specific problems or application domains. These applications include a sparse Gauss-Seidel solver [54, 56], a molecular dynamics simulation [53], a sparse matrix powers kernel [57, 58], and a Jacobi solver[41, 40, 39].

Each instantiation of full sparse tiling created to date has relied on specific properties of the given problem to simplify the tiling process. This dissertation work has developed a full sparse tiling algorithm that makes none of the simplifying assumptions of prior work. This section highlights some of the challenges of full sparse tiling in a general fashion. These challenges include the high algorithmic complexity of explicitly computing data dependency relations and avoiding race conditions caused by executing reduction loops in parallel. Further issues include the need to detect and respect data dependences between non-adjacent loops and the problem of efficiently finding initial partitions that result in improved locality. Our solutions to these issues are presented in the sections that follow.

3.2.1 Complexity of Data Dependency Computation

Any schedule generated by full sparse tiling must respect the flow, anti-, and output dependences present in the original code. However, explicitly creating the dependence relations in the general case can be computationally expensive. In this section, we step through

the process of finding the data dependence relations for the example given in Figure 3.1 to illustrate the complexity of the process.

Table 3.1 shows a tabular representation of the data access relations that are shown pictorially in Figure 3.1 as black arrows. Because the number of elements of a data space accessed by an iteration of a loop varies in the case of the Jacobi solver, the arity of the relation may vary. In this simple example, some iterations access a single data element and others access two elements. The general full sparse tiling algorithm developed in this research can handle this difference in arity while some other sparse tiling algorithms, such as that used in OP2 [38], cannot.

Once we have enumerated the data access relations, we must process them to find the data dependence relations representing the flow, anti-, and output dependences. The flow dependences for all pairs of loops L_x and L_y where $x < y$ are

$$\{\vec{i} \rightarrow \vec{j} \mid \vec{i} \in L_x \wedge \vec{j} \in L_y \wedge W_{L_x \rightarrow D_d}(\vec{i}) \cap R_{L_y \rightarrow D_d}(\vec{j}) \neq \emptyset\}.$$

The anti-dependences for all pairs of loops L_x and L_y where $x < y$ are

$$\{\vec{i} \rightarrow \vec{j} \mid \vec{i} \in L_x \wedge \vec{j} \in L_y \wedge R_{L_x \rightarrow D_d}(\vec{i}) \cap W_{L_y \rightarrow D_d}(\vec{j}) \neq \emptyset\}.$$

The output dependences for all pairs of loops L_x and L_y where $x < y$ are

$$\{\vec{i} \rightarrow \vec{j} \mid \vec{i} \in L_x \wedge \vec{j} \in L_y \wedge W_{L_x \rightarrow D_d}(\vec{i}) \cap W_{L_y \rightarrow D_d}(\vec{j}) \neq \emptyset\}.$$

In this simple example with only two loops, each dependence relation can only be between iterations of the two loops. Each loop has 17 data accesses, ten of which are reads and 7 of which are writes. To find the flow dependences requires that we compare each write of L_1 with each read of L_2. This results in 70 comparisons. Likewise, the anti-dependence relation also requires 70 comparisons and the output dependence relations requires $(|W_{L_1}| * |W_{L_2}|) = 49$. There are a total of 189 comparisons needed to find the three dependence relations. The relations are given in Table 3.2.

Within the context of general loop chains, finding the dependences has complexity $O((|R_{L_*}| * |W_{L_*}|) + (|W_{L_*}| * |R_{L_*}|) + (|W_{L_*}| * |W_{L_*}|))$. To put this in perspective, for a

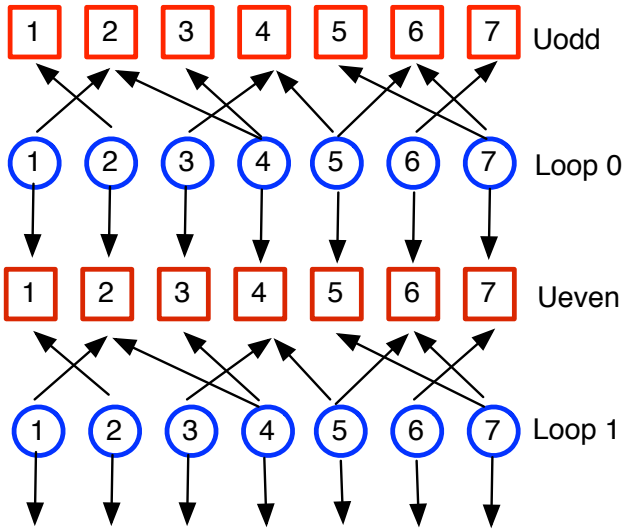


Figure 3.1: The data and iteration space view of the Jacobi code in Figure 1.1. This figure is a duplicate of Figure 1.2, duplicated here for convenience.

Table 3.1: Enumeration of the data access relations for the example given in Figure 1.2.

DAR on Loop 0	Elements	DAR on Loop 1	Elements
$R_{L_0 \to D_{Uodd}}(1)$	{2}	$R_{L_1 \to D_{Ueven}}(1)$	{2}
$W_{L_0 \to D_{Ueven}}(1)$	{1}	$W_{L_1 \to D_{Uodd}}(1)$	{1}
$R_{L_0 \to D_{Uodd}}(2)$	{1}	$R_{L_1 \to D_{Ueven}}(2)$	{1}
$W_{L_0 \to D_{Ueven}}(2)$	{2}	$W_{L_1 \to D_{Uodd}}(2)$	{2}
$R_{L_0 \to D_{Uodd}}(3)$	{4}	$R_{L_1 \to D_{Ueven}}(3)$	{4}
$W_{L_0 \to D_{Ueven}}(3)$	{3}	$W_{L_1 \to D_{Uodd}}(3)$	{3}
$R_{L_0 \to D_{Uodd}}(4)$	{2,3}	$R_{L_1 \to D_{Ueven}}(4)$	{2,3}
$W_{L_0 \to D_{Ueven}}(4)$	{4}	$W_{L_1 \to D_{Uodd}}(4)$	{4}
$R_{L_0 \to D_{Uodd}}(5)$	{4,6}	$R_{L_1 \to D_{Ueven}}(5)$	{4,6}
$W_{L_0 \to D_{Ueven}}(5)$	{5}	$W_{L_1 \to D_{Uodd}}(5)$	{5}
$R_{L_0 \to D_{Uodd}}(6)$	{7}	$R_{L_1 \to D_{Ueven}}(6)$	{7}
$W_{L_0 \to D_{Ueven}}(6)$	{6}	$W_{L_1 \to D_{Uodd}}(6)$	{6}
$R_{L_0 \to D_{Uodd}}(7)$	{5,6}	$R_{L_1 \to D_{Ueven}}(7)$	{5,6}
$W_{L_0 \to D_{Ueven}}(7)$	{7}	$W_{L_1 \to D_{Uodd}}(7)$	{7}

Table 3.2: Enumeration of the data dependence relations derived from the data access relations given in Table 3.1.

Relation (L_0 to L_1)	Dependency Type	Element Causing Dependency
[1] → [2]	Flow	Ueven[1]
[1] → [2]	Anti	Uodd[2]
[2] → [1]	Flow	Ueven[2]
[2] → [1]	Anti	Uodd[1]
[2] → [4]	Flow	Ueven[2]
[3] → [4]	Flow	Ueven[3]
[3] → [4]	Anti	Uodd[4]
[4] → [2]	Anti	Uodd[2]
[4] → [3]	Flow	Ueven[4]
[4] → [3]	Anti	Uodd[3]
[4] → [5]	Flow	Ueven[4]
[5] → [4]	Anti	Uodd[4]
[5] → [6]	Anti	Uodd[6]
[5] → [7]	Flow	Ueven[5]
[6] → [5]	Flow	Ueven[6]
[6] → [7]	Flow	Ueven[6]
[7] → [5]	Anti	Uodd[5]
[7] → [6]	Flow	Ueven[7]
[7] → [6]	Anti	Uodd[6]

typical sparse matrix used in this work by the Jacobi solver, there are on the order of 1×10^7 read access relations and 1×10^5 write access relations. The complexity of finding the data dependences is therefore $O(1 \times 10^{12})$, which is prohibitively expensive to compute at inspector time on most hardware available today.

Previously, special purpose full sparse tilers avoided this issue by using specific aspects of the application to reduce the complexity. With many of the computations that have been sparse tiled in the past, such as Jacobi, Gauss-Seidel, and the matrix powers kernel, inspecting the dependences between loops was equivalent to traversing index arrays. For example, in Jacobi, each iteration of a loop reads from a set of neighbors and writes to exactly one location. If the sparse matrix is stored in compressed sparse row format then the non-zero indices form a compact list of neighbor identifiers. By iterating over this list the dependence relation can be efficiently inspected. In these cases, the flow dependence relations can be reduced to

$$\{[i] \rightarrow [j] \mid i \in neighbors(j)\}$$

and the anti-dependence relations are reduced to

$$\{[i] \rightarrow [j] \mid j \in neighbors(i)\}.$$

The inspector can traverse the domain for the i or j iterations and inspect data dependences by looking in the neighbor set. This can be done in linear time. Also note that if a symmetric matrix is used, there is no need to find output dependences as they are fully contained within the transitive closure of the flow and anti-dependences. Several special purpose inspectors took advantage of this neighbor representation and dependence symmetry to reduce the complexity of finding the dependence relations. However, in the general case, these types of simplifications cannot be made, so a general methodology must address the dependence complexity issue in a more universal fashion.

3.2.2 Handling Parallel Reductions

The loop chain abstraction can describe loops that are either doall parallel or are reductions. The two types of loops can be scheduled in a similar manner when a serial tile schedule is being generated, as was the case with early full sparse tiling work. When these early tiling approaches were extended to support parallel execution [54], they relied on the fact that their specific sequences of loops did not contain reductions.

In the case of a general loop chain, that assumption cannot be made. When generating a parallel schedule in which multiple iterations of a single reduction loop may execute in parallel, it is necessary to avoid race conditions. These races occur when two or more processors read the same location in a reduction variable and modify it, oblivious to the changes made by the other processors. In this case, the changes made by some of the processors may be lost, resulting in data corruption. As was discussed in Section 1.4, full sparse tiling can generate schedules permitting multiple tiles to execute in parallel on a multicore machine. The partial ordering on tiles is expressed as a task graph. A general full sparse tiling approach must ensure that the task graph presents two or more tiles that all write to a single location in a reduction variable from executing in parallel.

3.2.3 Dependences Between Non-Adjacent Loops

One assumption made by previous single purpose full sparse tilers is that all data dependences exist between consecutive loops in the loop chain. For example, the first loop produces data that is consumed by the second loop and so forth. They do not support the case where, for example, loop L_2 produces data consumed by loop L_5. This simplifying assumption is made by the Jacobi, MolDyn, and OP2 full sparse tilers and holds true in all of those cases. When this assumption is true, the inspector can be simplified by searching pairwise between loops for data dependences, rather than having to search between all loops in the chain. This is a linear operation rather than a quadratic operation, so it greatly reduces algorithmic complexity.

To promote generality, there is nothing in the loop chain abstraction that restricts dependences to exist only between adjacent loops. Therefore, the full sparse tiler presented in this work does not rely on this assumption and has to directly address the $O(N^2)$ complexity in a different way.

3.2.4 Complexity of Creating the Initial Partitions

Much of the success of a full sparse tiling depends on the quality of the initial assignment of iterations from the seed iteration space to tiles. This assignment impacts the parallelism and spatial and temporal locality of the final tiling. To achieve a high quality initial iteration assignment, full sparse tilers attempt to assign iterations that access the same data to the same tile. The loop chain abstraction's data access relations express which iterations access which data items. By capturing all the iterations that access each particular data item, these access relations can used to create an *adjacency graph* with iterations as the vertices and edges between vertices that share data. The adjacency graph can then be partitioned using standard graph partitioning techniques to obtain initial tile assignments for the iterations in the selected loop. This technique results in tiles that have good temporal locality and data reuse.

In the general case, these adjacency graphs can be quite large, and the process of building them has complexity $O((NM)^2)$, where N is the number of iterations and M is the number of data elements. Previous approaches to full sparse tiling avoided the high asymptotic complexity of building the adjacency graph by either using different, already available, graphs in place of the adjacency graph. For example, the custom full sparse tiler for the Jacobi solver used the non-zero pattern in the sparse matrix in lieu of the full adjacency graph. Other full sparse tilers select a seed loop with a small constant number of data accesses per iteration and a limited number of iterations accessing each data item. For example, full sparse tilers working on irregular meshes [38] rely on the geometric properties of the mesh to limit complexity. A given mesh may have, for example, edges with only two connected vertices per edge and cells with only four edges.

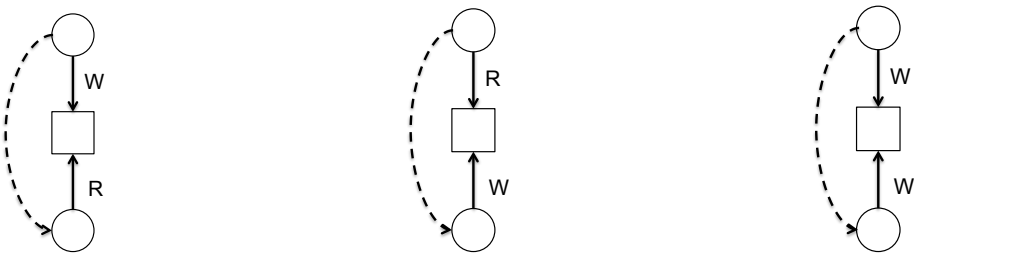
A general full sparse tiler must find some way of creating initial partitions in a universal and computationally tractable way. Some possible solutions include having the user cluster the input data before starting the tiling process or using a more efficient algorithm for creating the adjacency graph.

3.3 General Full Sparse Tiling Algorithm

The goal of full sparse tiling is to assign the iterations of each of the loops in a loop chain to one of a group of scheduling entities called *sparse tiles*. A sparse tile simply contains a list of iterations from each of the loops. When a sparse tile is executed, the iterations of the first loop are executed, then those from the next loop, and so on. All the iterations assigned to a sparse tile from one loop are completed before any assigned iterations of the next loop start execution. The loop chain definition forbids any chained loop from having ordering dependences between iterations of the same loop, so no ordering is necessary within a loop. In addition, sparse tiles are *atomic*, meaning that once input data is ready, the tile can execute to completion without any further communication or synchronization with other tiles. Tiles are numbered and, in a serial execution model, are executed in order, starting with tile number 0.

Iterations must be assigned to sparse tiles such that data dependences are respected. For example, if an iteration of loop 1 writes to memory address $A[5]$ and an iteration of loop 2 reads from that same address, the iteration that performs the read must execute after the iteration that does the write. This is a basic read after write (RAW) flow dependence. Figure 3.2 shows flow dependences, as well as write after read (WAR) anti-dependences and write after write (WAW) output dependences. The dashed arrows in Figure 3.2 show the ordering imposed on the execution of the iterations, shown as circles, due to the accesses to the shared data element, depicted as a square.

Full sparse tiling attempts to create tiling assignments that are not just legal, but that also exhibit good memory locality. It does this in two ways. First, it selects one loop to function as the *seed loop*. Iterations of this loop are assigned to sparse tiles based on the



(a) Flow Dependence. (b) Anti-Dependence (c) Output Dependence

Figure 3.2: Data dependences impose an ordering on iterations. Loop iterations are shown as circles and data elements are represented by squares. Accesses, shown as dark lines, are labeled to indicate whether they are reads (R) or writes (W).

data that they access. Different iterations that touch the same data are assigned to the same tile.

Next, iterations of other loops are added to tiles based on their sharing data with the iterations previously assigned from the seed loop. This process continues as iterations of more loops are added based on their having data accesses in common with iterations from any of the already tiled loops.

The grouping of iterations for locality is accomplished as a side effect of satisfying the data dependences. If an iteration of a loop reads from a location, it must be executed after any write to that location in any preceding loop. This means that the reading iteration must be assigned to a higher or equal numbered tile than any tile containing a write in a preceding loop. If we choose to assign the read to the tile containing the last write to that memory location, then both the write and the read will be within the same tile. As this process plays out for all the iterations, those iterations that share data accesses will naturally migrate into the same tile.

To parallelize the execution of full sparse tiles, the complete ordering given by tile number can be relaxed to a partial ordering based on data dependences between tiles. If two tiles have no data dependences, they can be executed in parallel. This partial ordering of tiles can be expressed as a *task graph* with tiles as vertices or tasks and ordering dependences as edges. Various task graph execution engines [3], including those developed as part of this dissertation work [39, 40], can then be used to execute the tasks in the task graph.

In the following sections, we more completely present the top level general full sparse tiling algorithm and then discuss each of the key elements in the general full sparse tiling process in greater detail. In Section 3.3.1, we explain the overall approach at a conceptual level and give an overview of the algorithm. We then cover how to initiate the full sparse tiling process in Section 3.3.2. In Section 3.3.4, we discuss sparse tile generation. Finally, in Section 3.3.5, we describe how a task graph is generated that captures inter-tile ordering dependences and enables parallel execution.

3.3.1 The Top Level Full Sparse Tiling Algorithm

The generalized full sparse tiling algorithm assigns iterations of the loops in the loop chain to sparse tiles. It then creates a partial ordering on the execution order of those sparse tiles. The overall full sparse tiling algorithm is given in Algorithm 1. The notation used throughout the algorithm is summarized in Table 3.3.

One change from earlier work in sparse tiling introduced in the general full sparse tiling algorithm is the Ψ access table. This data structure holds lists of all tiles that read or write a data element in a given loop. The list of tiles that read from data element $\vec{j}$ in data space D_d in the loop l is given by $\Psi_R(d, \vec{j}, l)$. Correspondingly, the list of tiles that write to data element $\vec{j}$ in data space D_d in the loop l is given by $\Psi_W(d, \vec{j}, l)$. The Ψ structure is discussed further in Section 3.3.3.

The generalized full sparse tiling algorithm starts by selecting one loop in the loop chain as the *seed space* or *seed loop*. This loop's iterations are assigned to tiles as detailed in Section 3.3.2 and those assignments are stored in the θ tiling function. Once these tile assignments are made, the *updatePsi* routine is called to capture the new tiling assignments and associate them with the data elements that are accessed by the tiles.

The algorithm then uses these initial tile assignments to bootstrap the rest of the tiling process. First the iterations of the loop immediately preceding the seed loop are assigned to tiles using the *backwardTile* routine. Each of these tile assignments is made such that all data dependences between each iteration of this loop and those of the seed space are

Table 3.3: A summary of the symbols used to represent input, state, and output data used in the Full Sparse Tiling process.

Symbol	Description
L	A sequence of loops $(L_0, L_1, \ldots, L_l, \ldots, L_{N-1})$.
l	An identifier for the l^{th} loop in the sequence.
L_l	The iteration space associated with the l^{th} loop in the sequence.
$\vec{i}$	A vector that identifies a specific iteration within a loop.
D	A set of data spaces $\{D_0, D_1, \ldots, D_d, \ldots, D_{M-1}\}$.
D_d	A specific data space. For example, D_0 may correspond to a data array X.
$\vec{j}$	A vector that identifies a specific element within a data space.
$R_{L_l \to D_d}(\vec{i})$	A relation between iterations and the data they read.
$W_{L_l \to D_d}(\vec{i})$	A relation between iterations and the data they write.
$\Psi_R(d, \vec{j}, l)$	The set of tiles that read a specific data element, $\vec{j}$,of D_d in loop l.
$\Psi_W(d, \vec{j}, l)$	The set of tiles that write a specific data element, $\vec{j}$, of D_d in loop l.
$\Psi_{FR}(d, \vec{j}, l)$	The first (lowest numbered) tile that reads a specific data element, $\vec{j}$, of D_d in loop l.
$\Psi_{LR}(d, \vec{j}, l)$	The last (highest numbered) tile that reads a specific data element, $\vec{j}$, of D_d in loop l.
$\Psi_{FW}(d, \vec{j}, l)$	The first (lowest numbered) tile that writes a specific data element, $\vec{j}$, of D_d in loop l.
$\Psi_{LW}(d, \vec{j}, l)$	The last (highest numbered) tile that writes a specific data element, $\vec{j}$, of D_d in loop l.
$\theta(l, \vec{i})$	The tile assignment for iteration $\vec{i}$ within loop l.

respected. The key idea behind backward tiling is to put iterations into tiles so that they execute before dependent iterations in later loops. Flow dependences can be preserved by putting iterations that write to a data location into tiles that execute before reads to that location in later loops. Similarly, anti-dependences can be honored by putting reads before writes to a particular memory location in a later loop. Likewise, output dependences are respected by putting writes to a data location into tiles such they execute before other writes to that location that occur in later loops. For example, if a data location is written by an iteration of this loop and later read by an iteration of the seed loop, the iteration must be assigned to a tile with an equal or lower tile number than the tile containing the seed loop read. This ensures that the flow dependence between these two iterations is satisfied. Information about what tile accesses each data element in each loop is readily available in the Ψ data structure, greatly simplifying this process. After each loop L_l is tiled and the tiling is stored in $\theta(L_l, *)$, the Ψ structure is updated using the *updatePsi* algorithm to reflect the tiles accessing each data element .

Note that *backwardTile* and *updatePsi* could be integrated into a single algorithm. Since *backwardTile* does not read from the Ψ table entries for the loop l that is currently being tiled and to which writes are actively occurring, there are no ordering or synchronization issues with combining the two steps. However, for performance reasons when the inspector code is parallelized, we perform the tiling process as two distinct phases.

This backward tiling process is then repeated for each other loop that precedes the seed space, moving from the loop immediately preceding the seed loop backward until the first loop is tiled. At this point, the algorithm changes direction and begins tiling the loops that succeed the seed loop. They are tiled using the *forwardTile* routine. Both the forward and backward tiling algorithms are presented in Section 3.3.4. The forward tiling routine closely resembles that used for backward tiling, but with one significant difference. The goal of the forward tiling algorithm is not to place iterations before later accesses but rather to place iterations into tiles *after* tiles containing earlier accesses. For example, if a data location is read by an iteration of a post-seed loop and was written by an iteration of an earlier loop,

```
1  GeneralizedFullSparseTile
   Input: Loop Chain LC=(L,D,R,W), seed loop index s
   Output: θ, 𝒢
   Data: Ψ
2
3  // Initialize all fields of Ψ to ⊤ or ∅
4
5  // Initialize the values in the tiling function, θ, to ⊤
6
7  // Assign iterations of the seed space to tiles
8  θ(L_s, ∗) = PartitionSeedSpace(L_s, R, W)
9  UpdatePsi(Ψ, s)
10
11 // Assign iterations from loops before the seed loop to tiles
12 foreach L_l in L_{s−1} to L_0 do
13 |   BackwardTile(L_l, l, R, W, θ, Ψ))
14 |   UpdatePsi(Ψ, s)
15 end foreach
16
17 // Assign iterations from loops after the seed loop to tiles
18 foreach L_l in L_{s+1} to L_{N−1} do
19 |   ForwardTile(L_l, l, R, W, θ, Ψ)
20 |   UpdatePsi(Ψ, s)
21 end foreach
22
23 // Create the parallel tile execution schedule as a task graph
24 𝒢 = BuildTaskGraph(Ψ)
25
26 return θ, 𝒢
27
```

Algorithm 1: The Generalized Full Sparse Tiling Algorithm

the iteration must be assigned to a tile with an equal or higher tile number than the tile containing the earlier read so as to respect the flow dependence. The forward tiling process begins with the loop immediately after the seed loop and continues loop by loop until the last loop in the loop chain has been tiled. This completes the assignment of loop iterations to tiles.

The tiling process consists of $|L| - 1$ calls to either *forwardTile* or *backwardTile* and a like number of calls to *updatePsi*. All of these have complexity $O(|R_{L_l}| + |W_{L_l}|)$, where $|R_{L_l}|$ and $|W_{L_l}|$ are the number of read and write access relations on loop l, respectively. The

seed space can be partitioned in different ways, but if we assume that that process also is $O(|R_l| + |W_l|)$, then tiling a loop chain has overall complexity

$$O(\sum_{l=0}^{|L|-1}(|R_{L_l}| + |W_{L_l}|)) = O(|R| + |W|)$$

and is therefore linear with respect to the total number of data accesses in the loop chain.

Once all iterations have been assigned to tiles, the tiles themselves are analyzed to see if the total tile ordering can be relaxed to a partial ordering. A task graph is used to capture the partial ordering on tile execution. The process examines each data item and determines if iterations in multiple tiles read or write the item. This is a straightforward process because all tiles that read or write a specific data element $\vec{j}$ in data space D_d during loop l can be pulled directly from $\Psi_{R|W}(d, \vec{j}, l)$. The task graph creation process needs no information beyond the Ψ structure. If two or more tiles read or write the data element, ordering edges are added to the task graph between those tiles to ensure that the iterations in the tiles respect all data dependences. This process is carried out by the *BuildTaskGraph* algorithm, which is discussed in Section 3.3.5.

Upon completion of the full sparse tiling algorithm, the θ function holds tiling assignments for all iterations in the loop chain. The $\mathcal{G}$ task graph holds the partial ordering on tile execution. This information can then be passed on to an executor that runs the tiled iterations using a parallel task graph execution engine.

3.3.2 Partitioning of the Seed Iteration Space

The entire full sparse tiling process begins by assigning iterations of the seed iteration space to tiles. Since the loop chain abstraction only allows loops without intraloop ordering constraints, any tiling of iterations is guaranteed to be valid. However, some tilings result in greater temporal and spatial locality. These tilings group together iterations that access data in common. If this concept is extended to iterations that access not just the same data element, but rather any element in the same cache line, even greater spatial locality may be achieved.

The quality of the seed space tiling has an impact on the quality of the tiling of the other loops as well. Recall from Section 3.3.1 that while backward and forward tiling, tiling assignments of other loops are all impacted by the assignments made for the seed iteration space. If iterations of the seed space that access common data are all assigned to the same tile, it is more likely that iterations of other loops that also access these data elements will be assigned to a given tile. Therefore, the quality of the seed space tiling is critical to the overall quality of the entire loop chain tiling.

There are many different approaches for creating high locality initial tilings. For a discussion of some of these possibilities, see Chapter 4. After researching different methods, we concluded that the lowest overall loop chain execution times are achieved when the user orders the data to align with the order of iteration execution. To preserve this correspondence between data and iteration ordering, the seed iterations should be assigned to tiles in a classic blocked fashion with a contiguous block of iterations assigned to tile 0, the next block of iterations assigned to tile 1, and so forth. This simple approach preserves the user's initial ordering when full sparse tiling.

Therefore, the algorithm *PartitionSeedSpace* referenced in Algorithm 1 is a blocked partitioner that assigns $\frac{|L_s|}{numtiles}$ iterations to each tile, where *numtiles* is the total number of tiles. It updates the tiling function values for the seed space, $\theta(L_s, *)$, to reflect these assignments.

3.3.3 Tracking Data Reads and Writes

During the development of the generalized full sparse tiling technique, we observed that much of the tiling and task graph building processes consist of placing an access to a data element either before or after other accesses to that same element. This is unsurprising because this is the essence of satisfying data dependences. We further observed that this information is needed only at the tile level of granularity. In other words, the algorithm only needs to know in which tile an access occurs, rather than the specific iteration that performs the access.

These observations led to the creation of the data access table, Ψ. The Ψ data structure holds lists of all tiles that read or write a data element in a given loop. The list of tiles that read from data element $\vec{j}$ in data space D_d in the loop l is given by $\Psi_R(d, \vec{j}, l)$. Correspondingly, the list of tiles that write to data element $\vec{j}$ in data space D_d in the loop l is given by $\Psi_W(d, \vec{j}, l)$.

Because during full sparse tiling we frequently need to know the first or last tile that reads or writes a data element in a specific loop, the Ψ structure also stores the tile number of the first and last tile to read from or write to each data element in a given loop. The first reading tile is referenced as $\Psi_{FR}(d, \vec{j}, l)$, the last reading tile as $\Psi_{LR}(d, \vec{j}, l)$, the first writing tile as $\Psi_{FW}(d, \vec{j}, l)$, and the last writing tile as $\Psi_{LW}(d, \vec{j}, l)$.

The process for updating $\Psi(*, *, l)$ is presented in Algorithm 2. It first clears all entries associated with the specified loop, L_l. It then visits each iteration $\vec{i}$ of the loop and examines the read data access relation $R_{L_l \to D_d(\vec{i})}$ on that iteration. The tile containing each read is added to $\Psi_R(d, \vec{j}, l)$. If a read was assigned to a tile earlier than the current value for $\Psi_{FR}(d, \vec{j}, l)$, that value is recorded as the new first reading tile. Similarly, if the tile number is greater than the previous last tile reading from the data element $\vec{j}$ in the current loop l, the entry for $\Psi_{LR}(d, \vec{j}, l)$ is changed to the new higher tile number. This process is repeated for the write relations in $W_{L_l \to D_d(\vec{i})}$, updating $\Psi_W(d, \vec{j}, l)$, $\Psi_{FW}(d, \vec{j}, l)$ and $\Psi_{LW}(d, \vec{j}, l)$ using similar logic.

The work needed to update the Ψ structure entries for a loop L_l depends on the number of accesses to data items in that loop. The asymptotic complexity is given by $O(|R_{L_l}| + |W_{L_l}|)$. Therefore, each application of the *updatePsi* algorithm completes in linear time.

3.3.4 Backward and Forward Tiling Algorithms

The backward and forward tiling algorithms are the key to achieving the threefold goals of full sparse tiling, namely validity, locality, and atomicity. The algorithms ensure that iterations are assigned to tiles such that all data dependences are satisfied. They also tile together iterations of different loops that access the same data, thereby improving temporal

UpdatePsi
Input: L, l, θ, R, W, Ψ
Output: Ψ

// Initialize the access tables of all data elements accessed
// by iterations of L_l
foreach $\vec{i} \in L_l$ **do**
  foreach $d \in D$ **do**
    foreach $\vec{j} \in R_{L_l \to D_d}(\vec{i}) \cup W_{L_l \to D_d}(\vec{i})$ **do**
      $\Psi_{FR}(d, \vec{j}, l) = \Psi_{LR}(d, \vec{j}, l) = \top$
      $\Psi_{FW}(d, \vec{j}, l) = \Psi_{LW}(d, \vec{j}, l) = \top$
      $\Psi_{R}(d, \vec{j}, l) = \Psi_{W}(d, \vec{j}, l) = \emptyset$
    end foreach
  end foreach
end foreach
// Update the access tables to reflect the current tiling
foreach $\vec{i} \in L_l$ **do**
  foreach $d \in D$ **do**
    foreach $\vec{j} \in R_{L_l \to D_d}(\vec{i})$ **do**
      $\Psi_{FR}(d, \vec{j}, l) = MIN(\Psi_{FR}(d, \vec{j}, l), \theta(l, \vec{i}))$
      $\Psi_{LR}(d, \vec{j}, l) = MAX(\Psi_{LR}(d, \vec{j}, l), \theta(l, \vec{i}))$
      $\Psi_{R}(d, \vec{j}, l) = \Psi_{R}(d, \vec{j}, l) \cup \theta(l, \vec{i})$
    end foreach
    foreach $\vec{j} \in W_{L_l \to D_d}(\vec{i})$ **do**
      $\Psi_{FW}(d, \vec{j}, l) = MIN(\Psi_{FW}(d, \vec{j}, l), \theta(l, \vec{i}))$
      $\Psi_{LW}(d, \vec{j}, l) = MAX(\Psi_{LW}(d, \vec{j}, l), \theta(l, \vec{i}))$
      $\Psi_{W}(d, \vec{j}, l) = \Psi_{W}(d, \vec{j}, l) \cup \theta(l, \vec{i})$
    end foreach
  end foreach
end foreach

return Ψ

Algorithm 2: Algorithm for Updating Ψ, the Data Element Access Table

```
1  BackwardTile
   Input: L, k, R, W, θ, Ψ
   Output: θ
2
3  Define: MIN(⊤,X) = X.
4  foreach i⃗ ∈ L_l do
5      foreach d ∈ D do
6          foreach j⃗ ∈ R_{L_l→D_d}(i⃗) do
7              foreach L_k ∈ {L_{l+1} to L_{N−1}} do
8                  // anti dependence - place iter so reads go before subsequent writes
9                  θ(l, i⃗) = MIN(θ(l, i⃗), Ψ_FW(d, j⃗, k))
10             end foreach
11         end foreach
12         foreach j⃗ ∈ W_{L_l→D_d}(i⃗) do
13             foreach L_k ∈ {L_{l+1} to L_{N−1}} do
14                 // flow dependence - place iter so writes go before subsequent reads
15                 θ(l, i⃗) = MIN(θ(l, i⃗), Ψ_FR(d, j⃗, k))
16                 // output dependence - place iter so writes go before subsequent writes
17                 θ(l, i⃗) = MIN(θ(l, i⃗), Ψ_FW(d, j⃗, k))
18             end foreach
19         end foreach
20     end foreach
21 end foreach
22
23
```

Algorithm 3: The Backward Tiling Algorithm

locality. Finally, they create tiles that are *atomic*, meaning that once initial external data dependences are satisfied, the tile can begin execution and can execute all its iterations completely without additional communication or synchronization.

The three goals are achieved simultaneously by the way the forward and backward tiling algorithms assign loop iterations to tiles. The backward tiling process is shown in Algorithm 3. We will discuss the backward algorithm first and then contrast it with the forward tiling algorithm.

The backward algorithm primarily focuses on placing iterations of a specified loop in a loop chain into tiles such that all three types of dependences, flow, anti, and output (see Figure 3.2), are satisfied. It does this by first assigning a loop iteration to the highest numbered tile. It then shifts the tile assignment lower and lower, to tiles that execute earlier and earlier, as required to meet data dependences.

For each iteration $\vec{i}$ in the specified loop L_l, the algorithm on line 6 steps through all the read access relations, $R_{L_l \to D_*}(\vec{i})$. Each of these is a relation, $[\vec{i}] \to [\vec{j}]$, stating that this iteration $\vec{i}$ accesses data element $\vec{j}$. In order to respect anti-dependences between this read to $\vec{j}$ and writes in iterations of later loops, this read must be placed before any such writes. To determine what tile meets this requirement, on line 9 we simply retrieve $\Psi_{FW}(d, \vec{j}, k)$ for each $k > l$. This tells us the first tile containing a write to this data element in the loop k. If that tile is less than the current tile assignment for iteration $\vec{i}$, that assignment is adjusted downward to match the writing tile. This is repeated for each loop k succeeding the current loop l, each time moving the iteration to an earlier tile or leaving the tile assignment untouched.

Satisfying flow and output dependences is accomplished in much the same way. The algorithm steps through all the write access relations $W_{L_l \to D_*}(\vec{i})$ on lines 12 through 18. To satisfy flow dependences, each of these writes must occur before a read in a later loop. Output dependences require the write to take place before a write in a later loop. Therefore, we retrieve $\Psi_{FR}(d, \vec{j}, k)$ on line 15 and $\Psi_{FW}(d, \vec{j}, k)$ for each loop $k > l$. If any of these tiles is lower numbered than the current tile assignment for iteration $\vec{i}$, the assignment is changed

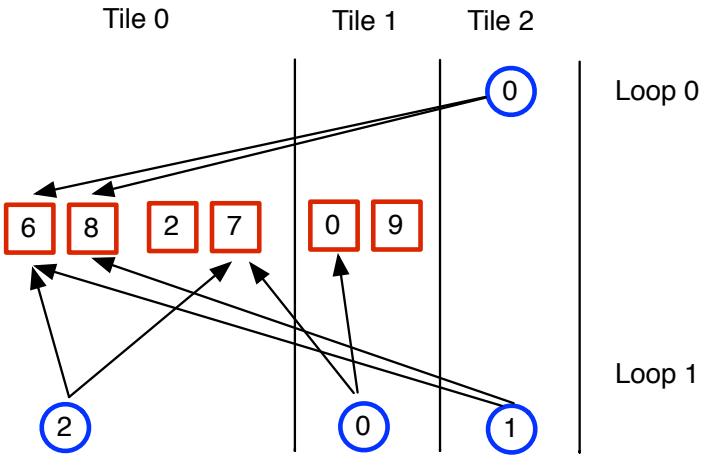
to match the lower tile number. In this way, all data dependences are respected. The final tile assignment is recorded in $\theta(l, \vec{i})$.

This process is illustrated by Figure 3.3, which pictorially shows the *backwardTile* algorithm tiling iterations of Loop 0. Loop 1 is the seed space and has already been tiled. A single data space, D_0 is present in the loop chain. Note that the order of the elements of the data space has been shuffled to reduce clutter in the diagram.

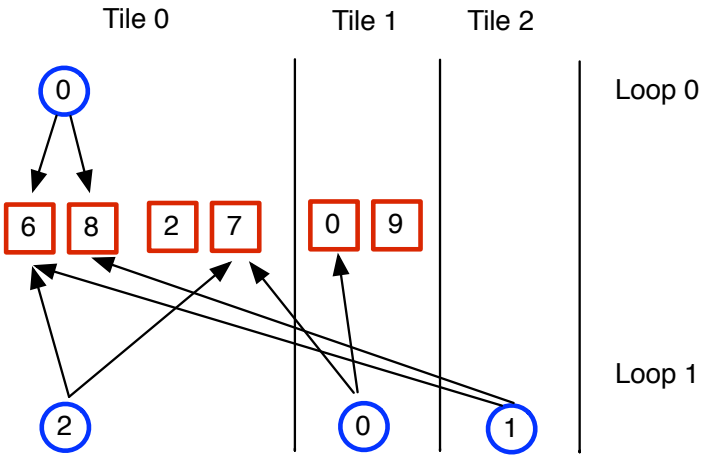
Before Algorithm 3 begins, Iteration 0 of loop 0 is initially assigned to $\top$ or the highest numbered tile. This initial placement is shown in Figure 3.3a. One line 6 of Algorithm 3, the read relations are examined. This iteration has no read relations, so lines 6 through 10 are not applicable. Iteration 0's write access relations are visited by the loop on line 12 and it is found that the iteration first writes to element 6. The $\Psi_{FR}(0, 6, 1)$ entry is queried on line 15. This is done for every loop after loop 0, which in this case is only loop 1. $\Psi_{FR}(0, 6, 1)$ is tile 0. As the min of $\top$ and 0 is 0, iteration 0 is moved from tile 2 to tile 0, as shown in Figure 3.3b.

Lines 12 through 18 are then repeated for element 8, which is also written by iteration 0. The value of $\Psi_{FR}(0, 8, 1)$ is tile 2, but since 2 is greater than 0, the iteration remains in tile 0. A simple check shows that elements 6 and 8 are now written in tile 0, before either iteration 1 or 2 of loop 1 reads them. Iteration 0 of loop 0 precedes iteration 2 because all iterations of a loop in a tile precede all iterations of later loops in that same tile. The iteration precedes iteration 1 of loop 1 because tile 0 has a lower tile number than tile 2, the tile that contains iteration 1.

The algorithm *ForwardTile*, presented as Algorithm 4, is the mirror image of *BackwardTile* and is used to tile loops after the seed loop. The key difference is that it first places an iteration into the first tile, tile 0. It then assigns an iteration to later and later tiles as necessary to satisfy dependences. In this way, it places iterations making data accesses such that they will execute after iterations that access the same data elements in earlier loops.



(a) Iteration 0 is initially assigned to tile 2



(b) Iteration 0 is forced into tile 0 to satisfy the flow dependence with iteration 2 of loop 1

Figure 3.3: A loop chain as it is being full sparse tiled. Loop 1 is the seed loop and iterations of loop 0 are being assigned to tiles by the *backwardTile* algorithm.

```
1  ForwardTile
   Input: L, l, R, W, θ, Ψ
   Output: θ
2
3  Define: MAX(⊤,X) = X.
4  foreach i⃗ ∈ L_l do
5      foreach d ∈ D do
6          foreach j⃗ ∈ R_{L_l→D_d}(i⃗) do
7              foreach L_k ∈ {L_0 to L_{l−1}} do
8                  //flow dependence
9                  θ(l, i⃗) = MAX(θ(l, i⃗), Ψ_LW(d, j⃗, k))
10             end foreach
11         end foreach
12         foreach j⃗ ∈ W_{L_l→D_d}(i⃗) do
13             foreach L_k ∈ {L_0 to L_{l−1}} do
14                 // anti dependence
15                 θ(l, i⃗) = MAX(θ(l, i⃗), Ψ_LR(d, j⃗, k))
16                 // output dependence
17                 θ(l, i⃗) = MAX(θ(l, i⃗), Ψ_LW(d, j⃗, k))
18             end foreach
19         end foreach
20     end foreach
21 end foreach
22
23
```

Algorithm 4: The Forward Tiling Algorithm

Observe that when backward tiling from the seed space, all loops after the seed loop will be untiled and will initially have their Ψ entries set to $\top$. This means that when backward tiling the loops before the seed space, no information about the tiling of loops after the seed space is available. Whenever comparing the current tile assignment with $\top$, the current assignment is preserved and the $\top$ value is ignored. At first glance, it may appear that data dependences may be violated, since iterations before the seed space must be tiled so as to precede iterations after the seed space, yet the tile placement of those iterations has yet to be made. Valid tilings, however, are actually produced under this scheme. When the loops after the seed space are later tiled using the *ForwardTile* algorithm, iterations will be appropriately placed *after* any iteration in any preceding loop. Because loop chains are a single sequence of loops without cycles, this approach always produces legal tilings.

The tiling assignments made by this algorithm also result in improved locality. An iteration is assigned to a tile such that it will either access an item also accessed by an earlier loop iteration in the tile or else access an item that will be accessed by an iteration of a later loop in the tile. While this assignment is made to satisfy data dependences, it also has the effect of improving temporal locality. One iteration will bring the data into the cache, and, if the reuse distance is small enough, those data will still be in a cache when the later iteration executes, resulting in improved performance.

The last quality of a sparse tile, that of atomicity, is also achieved by the tiling algorithm. For example, an iteration of a loop that writes to a data element will be scheduled either to the same tile as an iteration that reads from the element in a later loop or else placed in an earlier tile. When a tile begins executing, all the data accesses that must precede the execution of iterations in the tile either will have already executed or else will execute as part of this tile. By explicit construction and virtue of tiling by data dependence, it is never the case that a tile depends on data that is produced by a later tile.

```
1  BuildTaskGraph
   Input: L, l, D, Ψ, numtiles
   Output: G = (V, E)
2
3  V = {0, ..., numtiles − 1}
4  E = ∅
5  foreach l | L_l ∈ L_0 to L_{N−1} do
6      foreach d | D_d ∈ D_0 to D_{M−1} do
7          foreach j⃗ ∈ D_d do
8              // Reductions
9              E = E ∪ {[v] → [w] | v ∈ Ψ_W(d, j⃗, l) ∧ w ∈ Ψ_W(d, j⃗, l) ∧ v < w}
10             foreach k | L_k ∈ {L_{l−1}, ..., L_0} do
11                 // Flow dependences
12                 E = E ∪ {[v] → [w] | v = Ψ_LW(d, j⃗, k) ∧ w ∈ Ψ_FR(d, j⃗, l) ∧ v < w}
13             end foreach
14             foreach k | L_k ∈ {L_{l+1}, ..., L_{N−1}} do
15                 // Anti dependences
16                 E = E ∪ {[v] → [w] | v ∈ Ψ_LR(d, j⃗, l) ∧ w = Ψ_FW(d, j⃗, k) ∧ v < w}
17                 // Output dependences
18                 E = E ∪ {[v] → [w] | v = Ψ_LW(d, j⃗, l) ∧ w = Ψ_FW(d, j⃗, k) ∧ v < w}
19             end foreach
20         end foreach
21     end foreach
22 end foreach
23 return G
```

Algorithm 5: Task Graph Generation Algorithm

3.3.5 Task Graph Generation

The tiling process described in Chapter 3 assigns iterations of loops to tiles under the assumption that a lower numbered tile will execute completely before a higher numbered tile begins executing. In other words, executing tiles in numeric order by tile number is a valid schedule that satisfies all data dependences. However, it is often the case that tiles have no data dependences between them. In this case, the tiles can execute concurrently without violating data dependences.

Therefore, as part of the overall generalized full sparse tiling process, we examine the tile assignments and determine what tiles can be legally executed concurrently. This partial

order on tile execution is represented by a task graph with tiles as vertices and ordering dependences between tiles represented by edges.

The algorithm for building the task graph is shown in Algorithm 5. Note that it is able to determine the dependences between tiles using only the Ψ structure and never queries the θ tiling function. This highlights an advantage of tracking reads and writes in Ψ using tiles, rather than individual iterations.

The algorithm begins by creating a vertex in the graph for every tile. It then walks through the Ψ table, loop by loop and data element by data element. If at any point an entry in the Ψ table is set to $\top$, the comparison is recognized as invalid and no edge is added to the graph.

The first type of dependences considered are reductions. For each data element $\vec{j}$ and loop l, the algorithm examines $\Psi_W(d, \vec{j}, l)$. If two or more distinct tiles write to the data element in this loop, we must prevent a race condition by serializing the execution of the tiles. Therefore, the algorithm adds a task graph edge from the lower to the higher numbered tile for all tile pairs in $\Psi_W(d, \vec{j}, l)$.

Next, the cross-loop flow, anti, and output dependences are evaluated. If the data element under consideration was written in a loop prior to this loop and is read in this loop, a flow dependence may exist. To enforce this dependence, an edge is added between the last tile to write to the element in each previous loop and the first tile to read the element in this loop. To address anti-dependences, an edge is added to the graph between the last tile to read from the data element and the first tile to write to the element in each subsequent loop. To resolve output dependences, an edge is added to the graph between the last tile to write to this data item and the first tile to write to it in each later loop.

The process of building a task graph involves visiting each data element in the loop chain and doing Ψ table lookups $O(|L|^2)$ times for the loop to loop dependences. There is additional complexity of $O(|L| \times \sum_{d=0}^{d=|D|-1} |D_d| \times (number\ of\ tiles)^2)$ to handle reductions.

Therefore, the overall algorithmic complexity is

$$O((|L| \times \sum_{d=0}^{d=|D|-1} |D_d|) \times ((number\ of\ tiles)^2 + |L|))$$

with the dominant term usually being the number of data elements accessed.

3.4 Validity of the General Full Sparse Tiling Algorithm

The approach to satisfying data dependences taken by the generalized full sparse tiling algorithm is that of *correct by construction*. Given the specification that all data and reduction dependences must be satisfied, the algorithm places iterations into tiles such that these dependences are satisfied. Therefore, while the following assertions and justifications are not formal proofs of correctness, they are sufficient to convey concisely how the general full sparse tiling approach generates tilings that respect all dependences present in the loop chain.

Assertion: If there is a flow, anti, or output data dependence between iteration $\vec{i}$ in loop L_x and iteration $\vec{j}$ in loop L_y, then there is an edge in the task graph starting at $\theta(L_x, \vec{i})$ and ending at $\theta(L_y, \vec{j})$, $\theta(L_x, \vec{i}) \neq \theta(L_y, \vec{j})$.

Justification: If there is a dependence that ends at a particular iteration $\vec{j}$, then during the forward growth phase, the iteration $\vec{j}$ will be put in the same tile or a later tile than the iteration where the dependence starts, $\vec{i}$, due to the `MAX()` functions used at lines 9, 15, and 17 in the forward tiling algorithm given in Algorithm 4. Alternatively, during the backward growth phase, the iteration $\vec{i}$ will be placed in the same or an earlier tile than $\vec{j}$ due to the `MIN()` functions at lines 9,15, and 17 of Algorithm 3.

If the two iterations that share a dependence are in the same tile, two properties ensure proper ordering. First is the maintained sequencing between loops in a tile, meaning that all iterations of loop L_x assigned to a tile complete before the execution of any iterations of loop L_y, $x < y$, begin. The second property is the ordering independence of all iterations

within the same loop. This constraint is imposed by the loop chain abstraction. These two properties result in the dependence being satisfied by the schedule.

If the two iterations are assigned to different tiles, Algorithm 5 that builds the task graph will create an ordering edge from the earlier tile to the later tile by detecting that these two tiles are both writing to or one is writing to and the other is reading from the same data element.

Assertion: If there is a reduction dependence between iteration $\vec{i}$ and iteration $\vec{j}$, both in loop L_x, then there is an edge in the task graph starting at $\theta(L_x, \vec{i})$ and ending at $\theta(L_x, \vec{j})$, $\theta(L_x, \vec{i}) < \theta(L_x, \vec{j})$.

Justification: Reductions present a special case in that they contain dependences not between loops but within a single loop. If different iterations of the same loop read and write from the same memory location, they must either be placed in the same tile or else in two tiles that are prevented from executing at the same time. If not, a race condition will potentially exist. This race is prevented by line 9 of Algorithm 5, which places an edge from the lower numbered tile to the higher numbered tile for all pairs of tiles that write to the same memory element within the same loop.

3.5 Other Parallelization Approaches Related to Full Sparse Tiling

There is a large body of research related to full sparse tiling. In Section 2.6, we discussed prior work related to loop chains and their specification. In Section 3.1 we surveyed prior work on sparse tiling or related tiling efforts that directly led to the development of the current generalized full sparse tiling approach. In this section, we review other, related, solutions to the problem of automatic parallelization of sequences of loops.

Significant research on automatic parallelization of sequences of loops was done by Ravishankar et al. [51]. This work identifies *partitionable loops* and schedules these loops for execution on a distributed memory machine. It does not require a user to identify a loop

chain or declare which loops should be optimized. A hypergraph is generated with iterations from all loops as vertices and data elements as hyperedges. This hypergraph is partitioned to generate the iteration assignments to each processing node. In this approach, the tiles are not atomic and must communicate with other nodes after each loop, but locality across loops is improved. The system generates inspector and executor code using the ROSE source to source compiler system. This technique has been evaluated on a sparse conjugate gradient solver, similar to the sparse Jacobi solver used in our work, and achieved speedups of as much as 80 times on a 256 node machine.

Another approach to parallelization of loop sequences was developed by Basumallik and Eigenmann [7]. This work takes parallel loops identified by OpenMP pragmas and transforms them for execution on distributed memory clusters. The iterations executed on each node are reordered to maximize communication-computation overlap rather than to improve locality. Here again, the work assigned to each node is not atomic and communication is necessary during execution. This work is evaluated using the moldyn benchmark and a sparse conjugate gradient solver. This system produces code that performs comparably to hand coded MPI implementations at higher node counts.

The approach presented in this work differs from these techniques in three key ways. First, they produce code and data distributions for distribute memory systems. Therefore, a significant contribution of these efforts is related to data replication, distribution, and ongoing communication. The full sparse tiling work presented in this dissertation is focused solely on shared memory systems.

Second, these approaches generate a schedule in which each node or processing element executes its assigned iterations of one loop, then communicates a subset of its results to other partitions that are dependent on that data. After executing its iterations of a loop, each processing element potentially waits to receive data from other partitions. The full sparse tiling approach described here does not require any synchronization or communication during the execution of a tile due to the *atomicity* of the tile. Atomic tiles are better able to exploit the locality available across the sequence of loops.

Lastly, our approach optimizes sequences of loops that have been represented using the loop chain abstraction. These other approaches either rely on automatic identification of candidate loop sequences or else process loops identified by OpenMP pragmas. Either of these methods could be used to produce loop chain abstractions internally, as discussed in Section 2.3. Conversely, the systems overviewed here could be modified to work on the loop chain abstraction.

Chapter 4

Locality Considerations For Full Sparse Tiling

Locality of reference, or simply *locality*, is a property of a system or computer program [26]. This property indicates that future data accesses by the system can be inferred from an examination of the system's recent *reference trace* or list of memory locations previously referenced. Locality is sometimes divided into two kinds, temporal and spatial. A system exhibiting temporal locality will repeatedly access the same data element during a time interval. For example, the system may reference a sequence of data elements (1,6,3,1,4,6,5,1). Observe the repeated references to data elements 1 and 6. Systems with spatial locality will access data elements located near another in memory during a time interval. For example, consider a system that references data elements (1,7,2,9,3,4,10) sequentially. While no item is accessed repeatedly, within the reference trace are the patterns (1,2,3,4) and (7,9,10), runs that access nearby elements.

Most modern computer hardware has been architected to exploit the locality property through the use of a *cache*, or small, fast memory system that stores recently used data under the premise that the data or nearby data will be accessed soon [34]. Because the cache is small relative to main memory, it can only hold a limited number of recently used data elements. If the number of data elements accessed between two accesses to the same or nearby data, called the *reuse distance*, is less than the capacity of the cache, the later memory access will be serviced by the fast cache and will have lower access time than an access to main memory.

One of the primary objectives of general full sparse tiling is to reduce the execution time of a loop chain. It attempts to achieve this by placing repeated accesses to the same data element closer together in time by putting them into the same tile. This increases the

likelihood that subsequent accesses to the element will pull the data from lower latency cache memory. How this is accomplished and some challenges to improving locality through full sparse tiling are detailed in Section 4.1.

The full sparse tiling algorithm is also sensitive to how iterations of the seed space are assigned to tiles, so in Section 4.2 we discuss methods for improving seed space partitioning. Data reordering is a method for improving spatial locality that is largely orthogonal to full sparse tiling. At present, the generalized full sparse tiling algorithm does not alter the placement of data in memory. In Section 4.3, we present methods for reordering data that are complementary to full sparse tiling.

4.1 Interaction Between Locality and Full Sparse Tiling

Improving locality and converting that improved locality into better performance is at the core of generalized full sparse tiling. Iterations are assigned to tiles such that they share data with other iterations in the tile. This improves locality and will improve performance if the data accessed within a tile, known as its *data footprint*, is less than a cache level. In Section 4.1.2, we discuss what data contributes to the relevant data footprint of a full sparse tile. These footprints are not consistent across all tiles, so in Section 4.1.3, we discuss the statistical distributions of tile data footprints as generated by generalized full sparse tiling.

4.1.1 Iteration Placement To Improve Locality

The gFST algorithm increases temporal locality by placing references to the same data element into the same tile. This is a natural consequence of the way iterations are tiled. During the *backwardTile* step in the generalized full sparse tiling algorithm, an iteration that writes to a data element will be placed in the same tile as the earliest reader of that element in subsequent loops. An example of this behavior can be seen in Figure 4.1. In this example, iteration 0 of loop 0 is assigned to tile 0 in order to satisfy the flow dependence on element 6, read in iteration 2 of loop 1. This tiling assignment is driven by data dependences, but also results in placing the iteration into a tile where it shares at least one data element

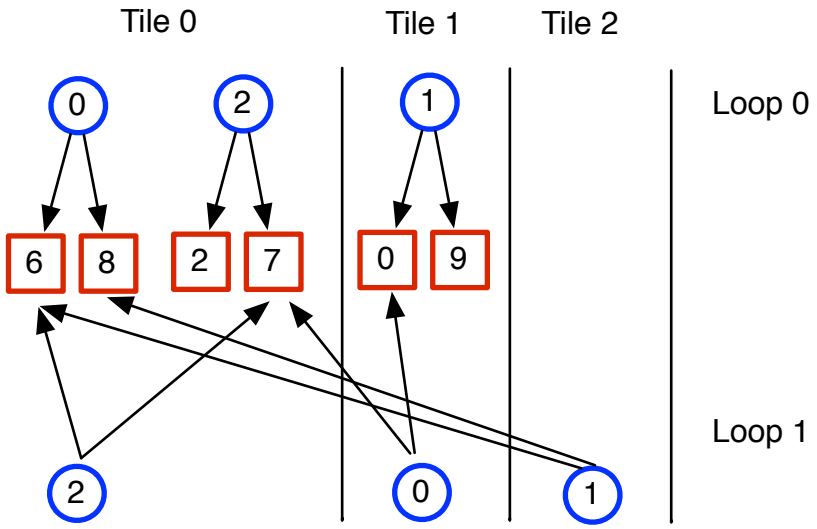


Figure 4.1: A full sparse tiling of a trivial loop chain. Loop 1 is the seed space. This example is a continuation of the example in Figure 3.3.

with an iteration of another loop. Under this tiling, iteration 0 of loop 0 and iteration 2 of loop 1 both access element 6 in tile 0. Likewise, iteration 2 of loop 0 and iteration 2 of loop 1 both access element 7 in tile 0 and iteration 1 of loop 0 and iteration 0 of loop 1 share element 0 within tile 1.

While full sparse tiling using the *backwardTile* and *forwardTile* algorithms can improve locality, an examination of Figure 4.1 reveals some apparent missed opportunities to place iterations sharing data into the same tile. For example, iteration 0 of loop 0 and iteration 1 of loop 1 share two elements, both 6 and 8. However, they are placed in two different tiles. This is because the placement of an iteration is dictated exclusively by its earliest dependence. In the case of iteration 0, the assignment is determined by the flow dependence with iteration 2 of loop 1. Any dependences with iterations assigned to later tiles, such as those with iteration 1 of loop 1 in tile 2, have no impact on the tiling assignment. This leads to a phenomenon called *locality dilution*, in which two or more tiles access the same data element, diluting the locality that would otherwise be contained within a single tile.

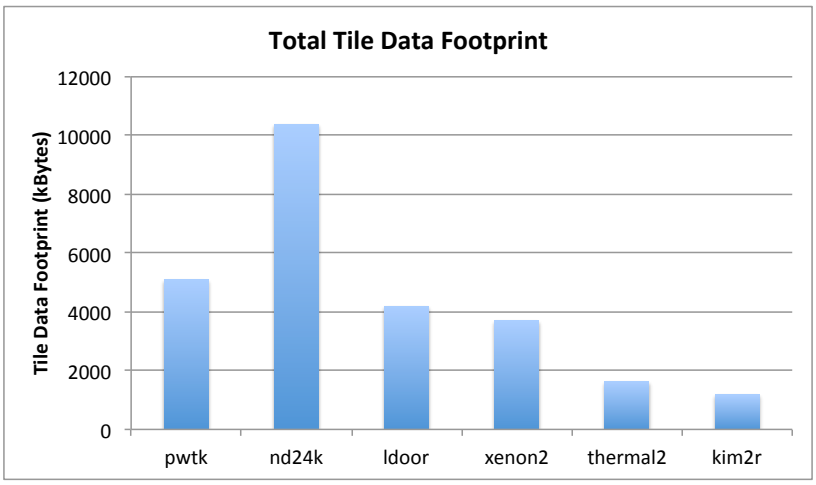


Figure 4.2: The total tile footprint of tiles yielding best performance. Results from the Jacobi solver with the optimal number of tiles are shown for 6 distinct sparse matrices.

4.1.2 Relationships Between Tile Footprints and Cache Sizes

We have found that minimum runtime is not achieved when the total tile footprint is equal to the size of the cache. Figure 4.2 shows the average tile footprint of optimally sized tiles from the Jacobi solver for six sparse matrices when run on a machine with a 32kB first level cache, a 256kB mid-level cache, and an 8 MB last level cache. Many of these tiles exceed the size of the mid level cache by more than an order of magnitude. The tile footprints for the different matrices also vary by more than a factor of eight from one another. The tile footprints are also not correlated with the total size of the working set for the Jacobi solver. For example, the ldoor matrix is 1.7 times larger than the nd24k matrix, yet its optimal tile size is less than half that of the nd24k matrix. This suggests that the optimal tile size is not strongly correlated with the total tile footprint.

Further examination of the memory usage of a Jacobi tile reveals that some data is accessed in a regular fashion while other data elements are accessed in an irregular fashion. If the tile footprint of only the irregular data is considered, a reasonable relationship between tile footprint and cache sizes emerges. Figure 4.3 shows the footprint of data accessed irregularly by an average tile for the same 6 sparse matrices used in Figure 4.2. Figure 4.3 shows that optimal performance is achieved when the irregularly accessed data fits within

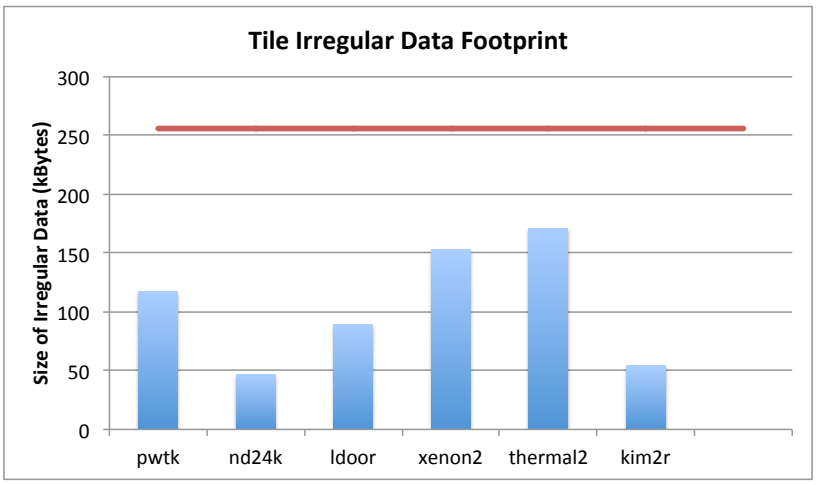
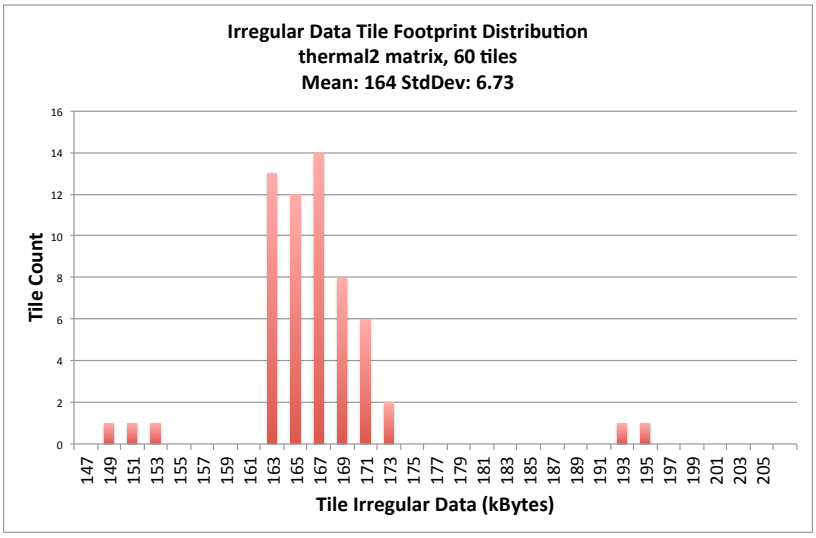


Figure 4.3: The size of irregular data per tile compared to the 256kB size of the mid level cache on a Xeon E3-1230 processor, as indicated by a red horizontal line. Results from the Jacobi solver with the optimal number of tiles are shown for 6 distinct sparse matrices.

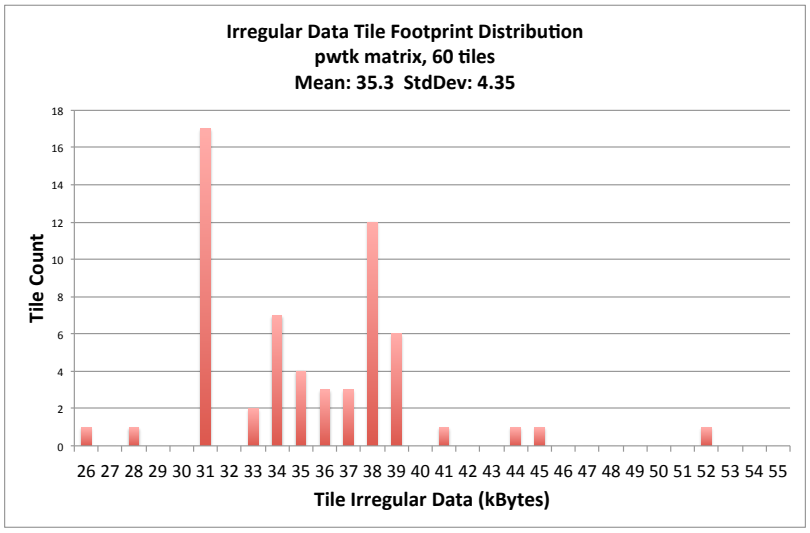
roughy one fifth to one half of the mid level cache. This ratio between irregular data accessed by a tile and the mid level cache size is reasonable because regularly accessed data will consume some portion of the cache, leaving less than the full cache available for irregular data.

4.1.3 Distributions of Tile Memory Footprints

Because the tile growth process assigns different numbers of iterations to each tile and because the amount of data accessed by each iteration potentially varies, the data footprint will also vary from tile to tile. This variation leads to a statistical distribution of tile data footprints. The histograms of tile irregular data footprints for four sparse matrices are given in Figures 4.4 and 4.5. None of the distributions neatly matches a common statistical distribution such as a normal, bimodal, or uniform distribution. Instead, they all have some clustering near the mean, but a significant number of outliers as well. The histogram in Figure 4.5b has a long tail to the right containing a significant number of tiles. For this reason, it is difficult to characterize tile size with a single metric such as mean or median. Often, the context when discussing tile footprint relates to finding a tile footprint that will fit in a particular cache. In this case, the cache size presents an upper bound on tile size.

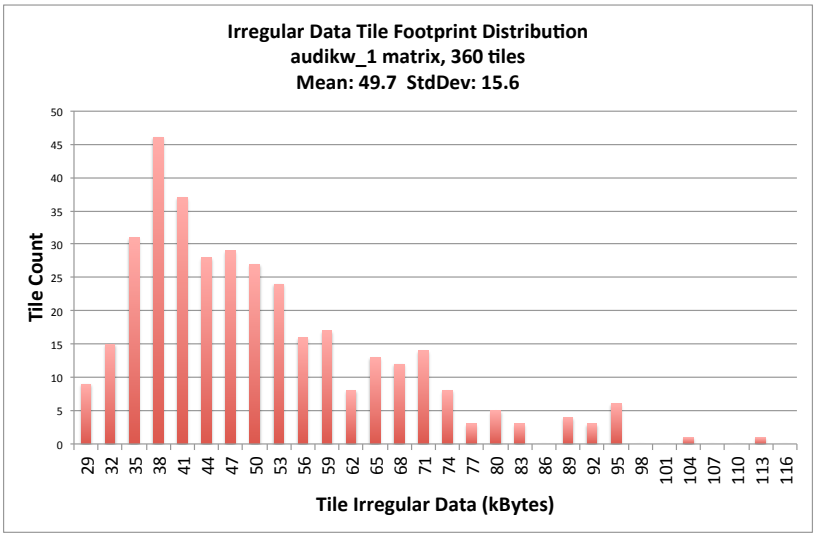


(a) Irregular data footprint distribution for the thermal2 matrix.

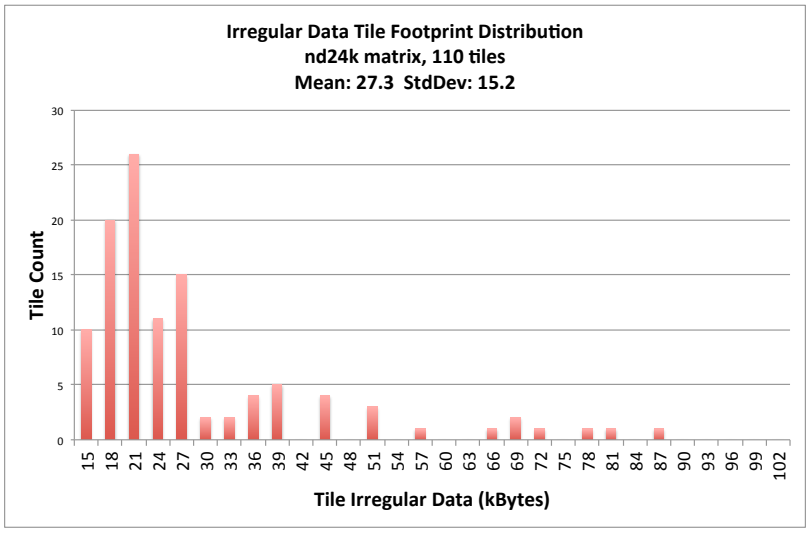


(b) Irregular data footprint distribution for the pwtk matrix.

Figure 4.4: Distributions of tile irregular data footprints. The tile count shown gave the best performance for the particular matrix.



(a) Irregular data footprint distribution for the audikw_1 matrix.



(b) Irregular data footprint distribution for the nd24k matrix.

Figure 4.5: Additional distributions of tile irregular data footprints. The tile count shown gave the best performance for the particular matrix.

If tiles are below the bound, they will fit in the cache. However, if there are other tradeoffs that reward larger tiles, there is a penalty to simply representing the distribution by its maximum. Doing so would place the majority of tile footprints far below the cache size. Placing the median tile footprint at the cache size would mean that half of the tiles fit in the cache and half do not. Observing histograms like those found in Figures 4.4 and 4.5, we notice that there tends to be either a tail to the right of the mean as seen in Figure 4.5 or else a small number of outliers to the far right of the histogram, as seen in Figure 4.4. To size the majority of tiles below a target size without creating many very small tiles, these large outlying tiles need to be ignored. The metric that captures most of the tiles, yet ignores the large outliers, is the 75th percentile. For these reasons, when capturing the distribution of tile footprints with a single number in this dissertation, the 75th percentile value is used.

4.2 Partitioning the Seed Space to Improve Temporal Locality

Recall that the *backwardTile* algorithm, Algorithm 3, and *forwardTile* algorithm, Algorithm 4, rely on the tile assignments of iterations from previously tiled loops to select tiles for iterations of the loop being tiled. Therefore, they are unable to make tile assignments until at least one loop has already been tiled. The initial loop, known as the *seed loop* or *seed iteration space*, must be tiled using a different approach.

When tiling the seed loop, improving temporal locality is the goal, just as it is when tiling other loops. To increase locality, iterations of the seed space that access data elements in common are assigned to the same tile. To accomplish this, two distinct steps are needed. First, information about what iterations access data in common is compiled. An adjacency hypergraph can be used to store this information. Then, based on that hypergraph, iterations of the seed loop that access data in common can be assigned to the same tile as much as possible. This can be achieved using a hypergraph partitioner.

The adjacency hypergraph is a hypergraph, $\mathcal{H} = (V, E)$, in which the vertices V are iterations of the seed iteration space and the hyperedges, E, represent data elements. Unlike

the ordered pairs that serve as edges of a simple graph, a hypergraph's hyperedges are subsets of V. These sets are unordered and can vary in cardinality.

The process of building the complete adjacency hypergraph for an iteration space is straightforward. The vertices are simply the iterations in the seed iteration space. The hyperedges are completely defined by the data access relations on the seed space. Building the adjacency hypergraph is a direct process of adding a pin, or vertex-to-hyperedge connection, for each member of the relation. The algorithmic complexity is $O(|R| + |W|)$.

Once the adjacency hypergraph is constructed, it can be used to assign seed space iterations to tiles. This is accomplished by *k-way partitioning* the adjacency hypergraph. The aim of k-way partitioning is to divide the vertices of the hypergraph into k distinct subsets such that a cost function is minimized. In the case of adjacency hypergraph partitioning, the cost function is the *edge cut*, or number of hyperedges that connect to vertices in different partitions. The number of partitions, k, is the number of tiles used by full sparse tiling.

Unfortunately, at the present time, there are no freely available parallel hypergraph partitioners for shared memory. There are several serial hypergraph partitioners, including PaToH [14, 13] and hMETIS [36]. As an alternative, all hypergraphs can be converted to graphs and then partitioned. However, the complexity of converting a hypergraph to a graph is $O(V \times E \times V)$, making conversion an unattractive option.

To partition the adjacency hypergraph, we developed a shared memory parallel hypergraph partitioner, hyperParCubed, that is based on our previous parallel graph partitioner, ParCubed [41]. HyperParCubed uses OpenMP to parallelize the process of assigning hypergraph vertices to partitions.

Before the hypergraph partitioning process begins, the data access relations on the seed iteration space are converted into a hypergraph with iterations as vertices and data elements as hyperedges. This hypergraph is then transposed to create a second hypergraph with data elements as the vertices and loop iterations as the hyperedges.

The hyperParCubed algorithm proceeds as follows. First, the seed iteration space is split into equally sized blocks of iterations, each of size $\frac{|L_s|}{t}$, where t is the number of threads

to be used for partitioning. Each thread takes the chunk of data edges assigned to it and processes each edge in turn by passing it to the per hyperedge partitioning algorithm. This algorithm then takes the hyperedge and adds it to a work list.

As long as there are hyperedges in the work list, the algorithm processes them as follows. First, it pops a hyperedge off the work list. For each vertex connected to the hyperedge, it considers whether or not to add the vertex to the current partition. The decision is based on whether the vertex has already been assigned to a partition, and if so, the size of that partition. If the vertex has not yet been assigned to a partition, it is added to the current partition and all hyperedges to which it is connected are added to the work list. Finding the connected hyperedges is straightforward using the transposed hypergraph. Otherwise, if the vertex in question is already in a partition, and that partition is small enough such that merging it with the current partition would not exceed the maximum partition size, the two partitions are merged. This process continues until either the work list is empty or the size of the current partition meets or exceeds the maximum size.

Commonly, the partitioning process produces more partitions than are desired, so an additional step is used to trim the number of partitions. During this *folding* step, the partitions are ordered by partition size from smallest to largest. If k partitions are desired, then the *adopting partition* P_k and *extra partition* P_{k-1} are merged, P_{k+1} and P_{k-2} are merged, and so forth. This combines increasingly smaller extra partitions with increasingly larger adopting partitions. If more than twice the desired number of partitions was originally found, the folding process functions in a modulo fashion, wrapping as needed.

Extra and adopting partitions are matched solely based on size. Due to this fact, unconnected partitions can be created. Also note that during folding, the maximum partition size is ignored, so partitions that exceed the desired size can be produced.

The hyperParCubed partitioning approach creates partitions in which iterations in a partition share at least one data element with another iteration in the partition. This improves temporal locality when executing iterations of the seed iteration space.

4.3 Data Reordering and Generalized Full Sparse Tiling

In Figure 4.1, one can observed that the spatial locality within the tiles can be improved. Looking at tile 0, we see it accesses elements {2,6,7,8}. If element 2 were moved to location 5, all the accesses would be contiguous as {5,6,7,8}. Similarly, tile 1 accesses elements {0,9}. If element 9 were moved to location 1, tile 1 would access contiguous elements as {0,1}. Because the generalized full sparse tiling algorithm does not reorder data, it is not able to improve spatial locality in this way.

Improving the spatial locality of the data access pattern of a loop chain is not an intrinsic feature of the full sparse tiling process. Two iterations that access data elements that are stored in adjacent memory locations are not automatically placed into the same tile, as they do not share a producer-consumer relationship. Compilers cannot reorder the data in many cases because data is being accessed irregularly based on other data read at run time.

Ding and Kennedy have done work on dynamic data reordering using an inspector/executor approach. In [28], they apply several reordering techniques to data from the moldyn benchmark. Wood studied the benefit of applying partitioner based reordering [57] to data used in a sparse matrix powers kernel application. This prior work shows that there can be a clear benefit to data reordering in the types of applications that also benefit from full sparse tiling.

As part of this dissertation, we examined the impact data reordering had on the Jacobi solver. In particular, we wanted to know if data reordering techniques used to improve performance of the blocked Jacobi solver would also benefit a full sparse tiled version. For this study, we reordered 6 sparse matrices from the University of Florida Matrix Market [24]. For each matrix, the ReorderMM tool from [57] was used to reorder the data. This tool takes a partition based reordering approach in which data is broken into partitions. Data is then reordered such that elements in a partition are placed together in the data ordering. We created partition sizes ranging from 32 kilobytes of data up to 16 megabytes in steps

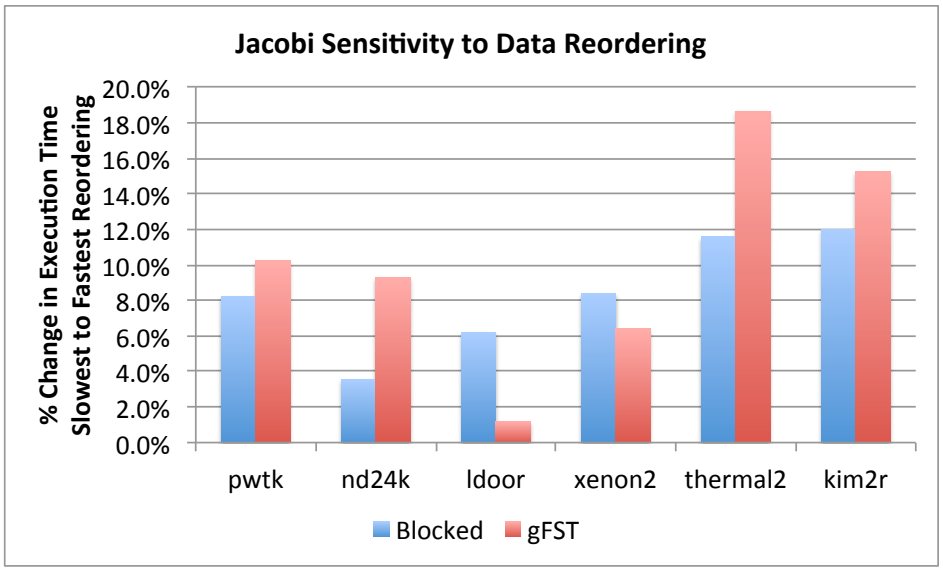


Figure 4.6: Sensitivity to data ordering of both the blocked and full sparse tiled Jacobi solvers. The percent change shown is between the ordering yielding the highest performance and the ordering giving the lowest performance.

increasing by a power of two. These 10 matrix reorderings, along with the original data ordering, were sent to both the simple blocked version of the Jacobi solver as well as the full sparse tiled version. Figure 4.6 shows the percent change in execution time between the data orderings delivering the best and worst performance. The performance varies between 1.2% and 18.7% and on average the delta was 8.3% and 10.2% for the blocked and full sparse tiled solvers, respectively. This indicates that data reordering can be a significant factor in performance. Also note that for some matrices, the blocked solver showed greater sensitivity to data reordering than the gFST solver, while for others the opposite was true.

Figure 4.7 shows the partition sizes that resulted in the best performance for each of the matrices for the blocked and full sparse tiled Jacobi implementations. In most cases, the same data reordering that lead to the best performance for the blocked solver also resulted in the best performance for the full sparse tiled solver. In cases where the two solvers performed best with different data reordering, the difference in reorderings was never more than a single step in partition size. This indicates that data reordering done to improve the performance of a blocked implementation also benefits the same loop chain after full sparse tiling.

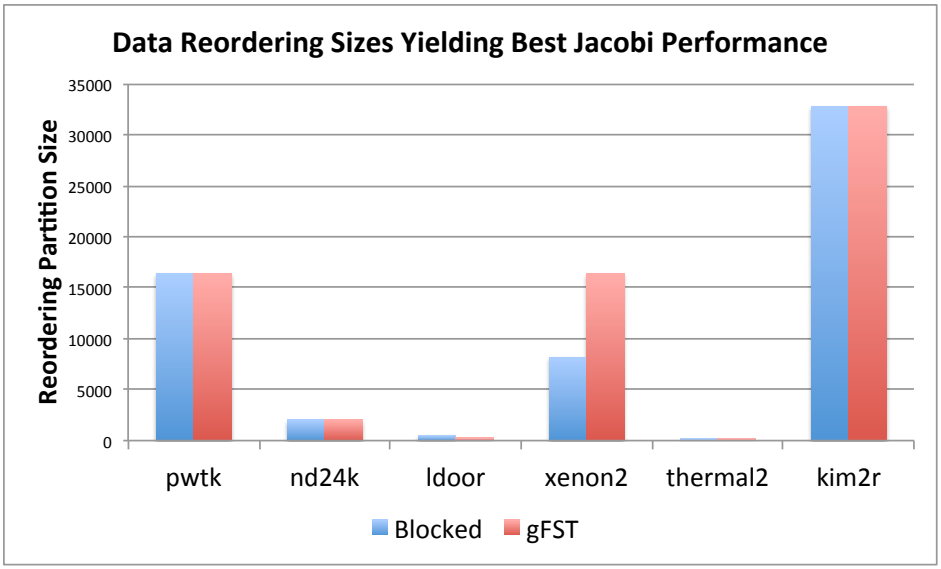


Figure 4.7: Partition sizes resulting in the best performance for the blocked and full sparse tiled Jacobi solvers. In general, both solvers benefit from similar data orderings.

Chapter 5

Parallelism Considerations for Full Sparse Tiling

One of the major aims of the generalized full sparse tiling approach is to parallelize the execution of a loop chain. After tiling together iterations to improve locality, the generalized full sparse tiling algorithm concludes by building a task graph that expresses the parallelism available in the tiled schedule. If there is sufficient parallelism in the task graph to fully utilize the available hardware resources, good performance will be achieved.

During the generalized full sparse tiling process, iterations of the seed space are assigned to tiles so that they share data accesses. This is achieved through partitioning, as described in Section 4.2. The tile numbers of these initial partitions are treated as a partial ordering on tile execution. Tiles with lower tile numbers execute before tiles with higher tile numbers. If tile numbers are assigned to the initial partitions such that consecutively numbered tiles share data dependences, the task graph will be highly serialized. In Section 5.1, we discuss a method for mitigating this effect to significantly increase the parallelism of the task graph.

It is important to realize that simple metrics such as average parallelism are inadequate to capture the intricacies of a task graph. In Section 5.2, we discuss different metrics and techniques for evaluating the parallelism of a task graph. Using these metrics, in Section 5.3, we present a study of how much median parallelism is needed to achieve high performance on multicore machines.

5.1 Coloring Seed Partitions To Improve Parallelism

When executing a full sparse tiled loop chain, the amount of parallelism available is dictated by the shape of the task graph. Graphs that are wide and shallow, rather than narrow and deep, better utilize multicore hardware resources. The generalized full sparse

tiling approach numbers seed partitions in a fashion that promotes the development of task graphs with this broader shape.

During the seed space partitioning process, described in Section 4.2, the iterations of the seed space are assigned to tiles. These tiles are assigned an ordering by the partitioner such that tiles that share data are frequently assigned adjacent numbers. A consequence of this ordering is that, in many cases, two consecutively numbered tiles contain iterations that have a data dependence between them. This results in the two tiles being strictly ordered. This leads to a task graph that is narrow and deep with little parallelism.

To resolve this issue, a different tile numbering scheme is used in generalized full sparse tiling. This approach is drawn from work presented in [54]. Rather than assign consecutive numbers to tiles that share data, tiles are assigned numbers such that, to the degree possible, consecutively numbered tiles do not share data. This is accomplished by creating a *partition graph* [54]. This graph has the initial tiles as the vertices and has edges between vertices that share data. This graph is then colored using a standard graph coloring algorithm such that adjacent tiles are assigned different colors [22]. Tiles that share a color will not share data. Tiles of the first color are then assigned consecutive tile numbers. This is repeated in turn for each of the colors. Under this scheme, the tile numbering contains runs of same-colored tiles that have no dependences. The length of these runs, or equivalently, the cardinality of a color set, is an indicator of the amount of parallelism that will be present in a level set of the final task graph.

While coloring the graph is a fast and straightforward process, building up the partition graph is algorithmically complex. Building the partition graph is $O(|L_s|^2)$, the square of the number of iterations in the seed iteration space. This complexity means that generating the partition graph can take longer than creating the initial seed partitions and coloring the graph combined.

Note that this coloring step is done purely to increase the parallelism of the final task graph. It is not necessary to guarantee the correctness of the parallel task graph. This is in contrast to the approach used by the OP2 full sparse tiling researchers [38]. Their

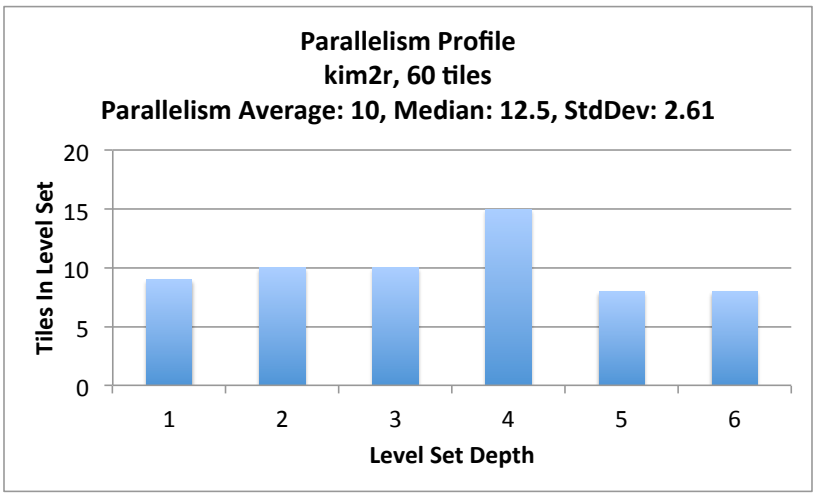


Figure 5.1: Parallel profile for a 60 tile full sparse tiling of the Jacobi solver on the kim2r sparse matrix.

methodology relies completely on graph coloring to discover what tiles can execute in parallel. This is similar to the multi-coloring technique used by Barrett et al. in [5]. The work presented in this dissertation uses Algorithm 5 to determine the dependences between tiles.

This distinction is important because it allows us to color an approximation of the true partition graph without encountering any correctness issues. As the hypergraph partitioner is creating the initial seed space partitions, it keeps track of each time it encounters an iteration that would be assigned to a tile but cannot because it has already been assigned to a different tile. These conflicts indicate that two different tiles are accessing the same data element and so the conflicts are logged as they are discovered. After partitioning is complete, the logged conflicts can be converted into an approximate partition graph in $O(N)$ time, where N is the number of conflicts. This reduction in graph construction complexity allows graph coloring for improved parallelism to be completed in a timely manner.

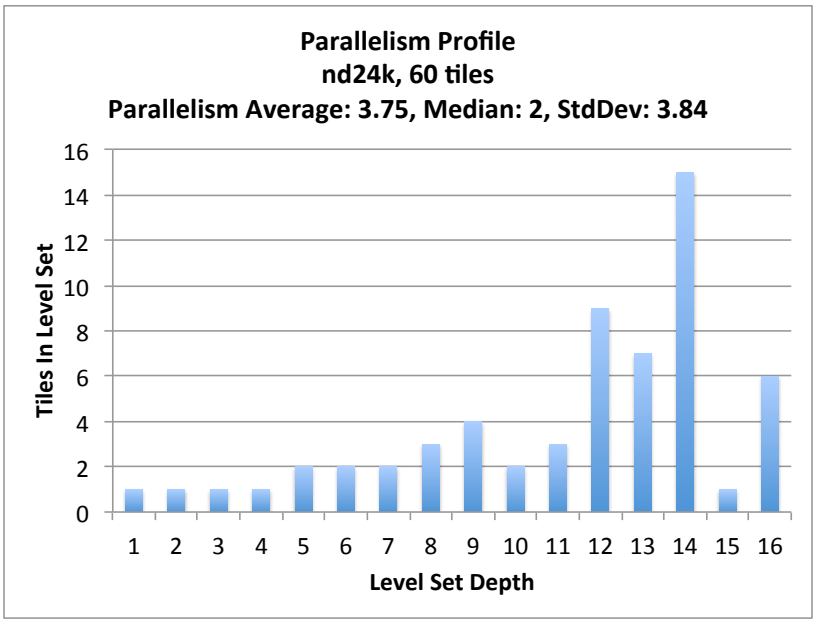


Figure 5.2: Parallel profile for a 60 tile full sparse tiling of the Jacobi solver on the nd24k sparse matrix.

5.2 Issues with Measuring and Controlling Task Graph Parallelism

5.2.1 Statistics for Measuring Parallelism

When striving to understand the parallelism available in a task graph, it is often necessary to understand how the available parallelism varies over the execution of the task graph. This can be illustrated using a *parallel profile*. A parallel profile breaks the task graph into *level sets*. A level set consists of all the tasks that can execute at a specific depth in the task graph. Level set n contains those tasks whose predecessors are all elements of the union of level sets 0 through $n-1$. For example, the first level set contains those tasks in entry nodes of the task graph. The second level set contains those tasks whose predecessors are a subset of the tasks in the first level set, and so on for deeper level sets.

Observe the parallel profile shown in Figure 5.1. This is the parallel profile of a 60 tile full sparse tiling of the Jacobi solver on the kim2r sparse matrix. It has an average parallelism of 10 and a median parallelism of 10 and has good parallelism across all level sets. The average

and median parallelism in this case are both a good indicator of how many processors could be effectively used.

Contrast that parallel profile with Figure 5.2 that shows the parallel profile for a 60 tile full sparse tiling of the Jacobi solver on the nd24k sparse matrix. The task graph has an average parallelism of 3.75. Considering only the average parallelism, it would seem that the task graph could consistently utilize the cores in a 4 core system. However, the parallel profile shows that only 5 of the 16 level sets contain 4 or more tasks. The large level sets found at depths 12 through 14 and 16 lift the average and obscure the fact that during the majority of the task graph level sets, there is not enough parallelism to keep more than two cores engaged. The median parallelism value of 2 is a better indicator of how much parallelism is present in the task graph. However, there is still a benefit to provisioning more cores than the median or mean parallelism indicates. Figure 5.2 shows that, for one quarter of the level sets, there is enough parallelism to benefit from the use of additional cores.

The incompleteness of the mean or median width of the task graph to fully represent parallelism highlights the limitations of using a single statistic to represent a distribution or function. When summarizing parallelism, in this work, we use the median parallelism. However, we acknowledge that significant information is lost in doing so and rely on full parallel profiles when median or mean values are inadequate.

5.2.2 Using Tile Count to Control Parallelism

Techniques that attempt to control task graph parallelism through the use of tile count rely on knowing the relationship between tile count and median parallelism. Using this relationship, a given median parallelism target can be reached by setting the tile count to the corresponding value. Unfortunately, there is not a simple or well-defined relationship between tile count and parallelism. In some cases, the relationship is roughly linear. For example, Figure 5.3 shows a well-behaved relationship between tile count and median parallelism. There is nearly a linear relationship with a slope of approximately 0.04. This indicates that increasing the tile count by 25 tiles will in general increase the median parallelism by one.

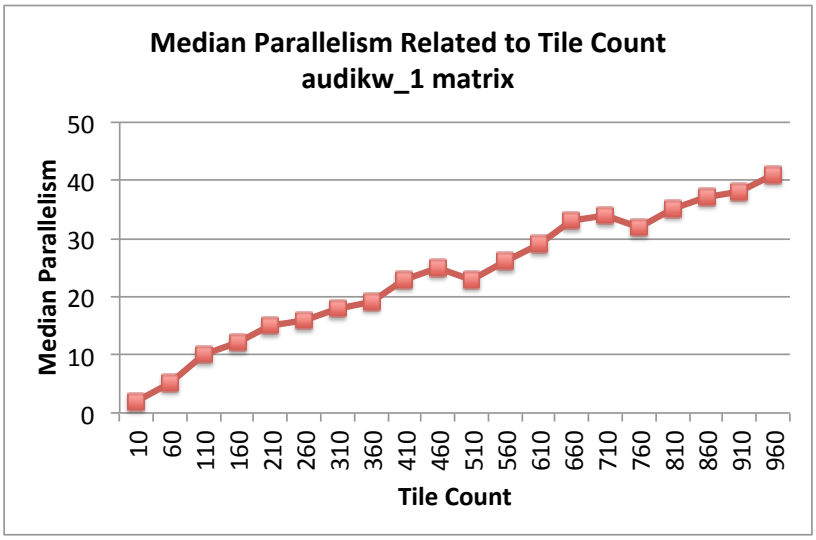


Figure 5.3: Relationship between median parallelism and tile count for a tiling of the audikw_1 matrix

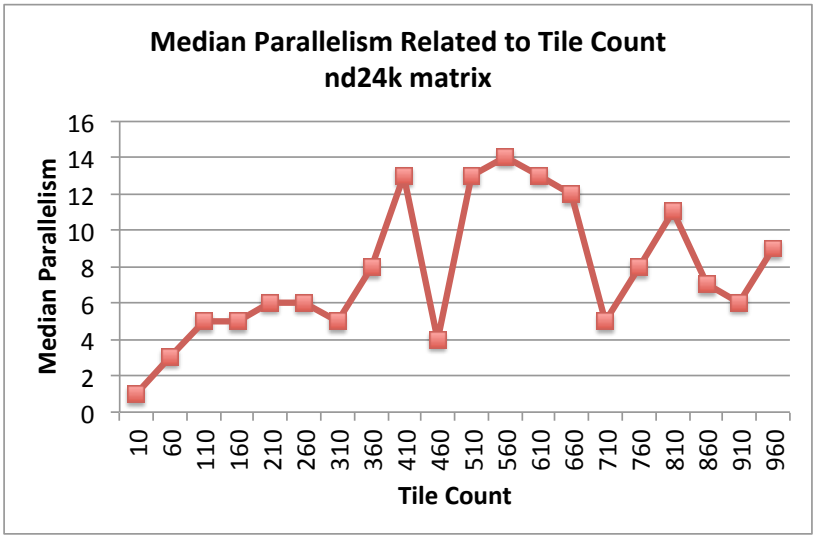


Figure 5.4: Relationship between median parallelism and tile count for a tiling of the nd24k matrix.

However, this slope is specific to the Jacobi solver on this particular sparse matrix and does not generalize to other applications or matrices. It is not clear what factors influence the relationship between tile count and median parallelism. It appears that the data access pattern of a tile, among potentially other factors, impacts how many tiles are needed, but what specific aspects of that pattern are unclear.

Even in this well-behaved case, note that the line in Figure 5.3 is not monotonically increasing. Increasing the tile count from 460 to 510 tiles actually results in a reduction of median parallelism. There is a similar dip in median parallelism going to 760 tiles from 710 tiles.

In other cases, the relationship is not linear at all. In Figure 5.4, the relationship between median parallelism and tile count is shown for the nd24k sparse matrix. Here the median parallelism does not follow a linear trend and both increases and decreases with increased tile count. Depending on the type of optimization being performed, such as gradient descent, functions like that shown in Figure 5.4 can cause the optimizer to become trapped in local minima rather than to find the global minimum.

5.3 Determining the Optimal Amount of Parallelism

Determining the optimal amount of parallelism is an important aspect of executing a task graph. Intuitively, the parallelism present should equal or exceed the number of processing elements. However, increasing parallelism usually entails increasing the number of tiles. This in turn increases scheduling overhead and reduces the size of each tile, potentially shifting the tile size away from a desired memory footprint such as a cache size. Therefore, increasing the tile count to increase the parallelism comes at a cost.

To determine how much parallelism results in optimal performance, we conducted experiments on three multicore machines. The first machine has a single four core Xeon E3123 processor. The second machine has eight cores using 2 Intel Xeon E5450 quad-core processors while the last is a 24 core node on the ISTeC Cray XT6m supercomputer using two

AMD Magny-Cours 12 core processors. These experiments sweep the tile count from 10 to 960 and the core count from one to the maximum number of cores.

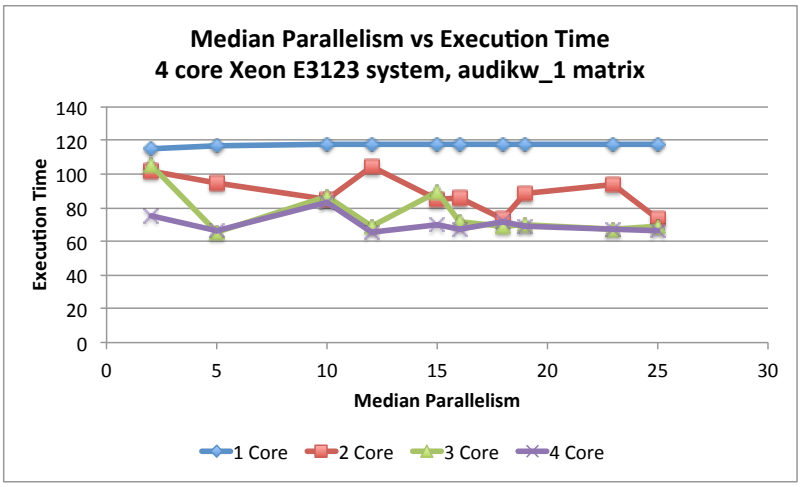
The results of these experiments are shown in Figures 5.5 through 5.11. Each of the graphs plots the execution time of 1000 convergence iterations of the Jacobi solver against the median parallelism in the task graph. This same experiment is conducted for seven different sparse matrices on each of the three hardware platforms. Figure 5.6 shows the basic pattern seen across the data sets. The execution time of the solver initially is high, but quickly drops as median parallelism is slightly increased. This is true for single core execution as well as parallel execution and is due to the tile size shrinking such that it fits in the cache. This occurs at a median parallelism between 15 and 20 for the pwtk matrix. As median parallelism is increased beyond this point, execution time decreases slightly until median parallelism reaches approximately 40. Beyond this point, execution time is flat to slightly higher as parallelism is increased.

Surprisingly, this same pattern holds for almost all the matrices and machines. Across Figures 5.5 through 5.11, the lowest parallel execution time always occurs with a median parallelism between 30 and 40. This is true regardless of the number of cores, size of input data, total machine bandwidth, or any other machine or matrix characteristic. In particular, it is independent of the number of tiles needed to reach that degree of parallelism. The one exception to this rule is the nd24k matrix, whose results are given in Figure 5.7. Its results vary widely with the degree of parallelism. Figure 5.4 shows the relationship between tile count and parallelism and shows that the same median parallelism is achieved with different tile counts. This leads to multiple execution times assigned to the same median parallelism. Additional variability stems from load imbalance and is discussed in Section 6.4. This same phenomenon is seen to a lesser degree in Figures 5.5 and 5.10.

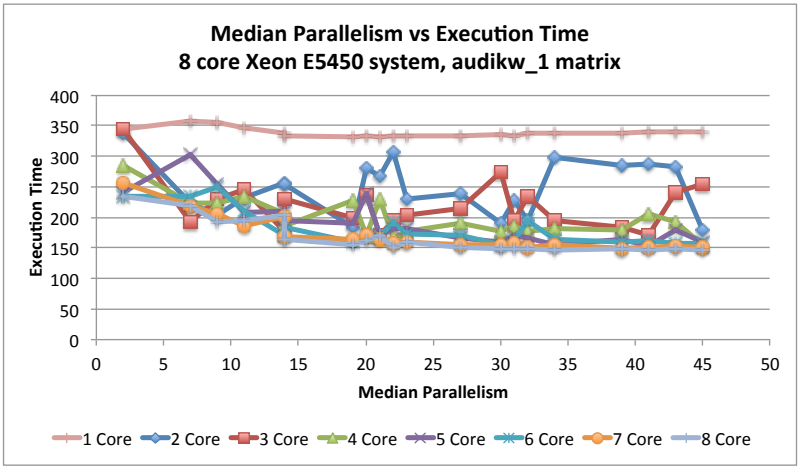
Within the range of tile counts used for these experiments, nd24k never reached a median parallelism higher than 14. Because its parallelism increases so slowly with tile count, it is the one case where the basic tenant that parallelism should exceed core count can be seen. Figure 5.7a shows that execution time decreases as median parallelism increases to four, the

number of cores present in that system. In Figures 5.7b and 5.7c, the trend is less clear due to noise, but increasing parallelism to match the core count again improves performance.

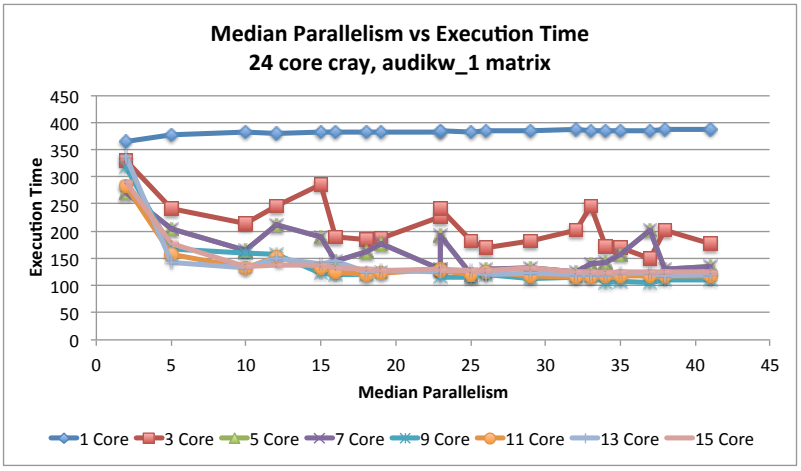
This significant finding that parallel performance requires a fixed range of median parallelism greatly simplifies the process of selecting the correct amount of median parallelism. Any optimization or parallelism selection algorithm need only drive the parallelism into the 30 to 40 range to achieve good results. Within that range, additional tuning may be accomplished through trial and error or exhaustive search to achieve peak results.



(a) Results from a Xeon E3123 4 core machine.

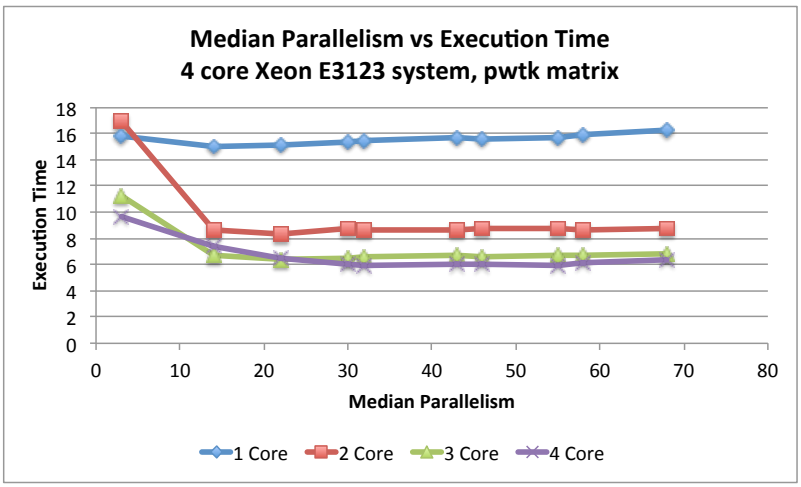


(b) Results from a Xeon E5450 8 core machine.

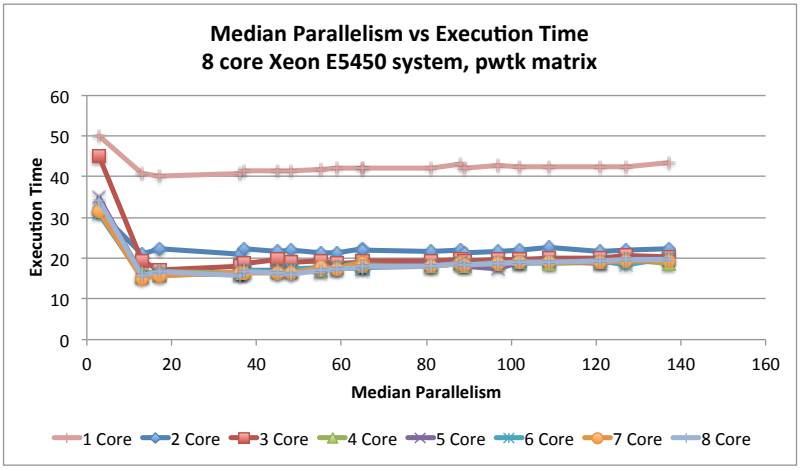


(c) Results from a Cray 24 core node

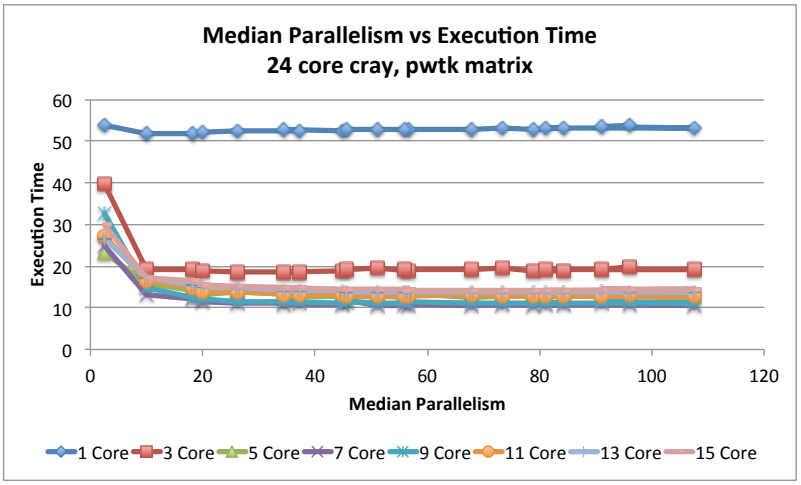
Figure 5.5: Median parallelism yielding the best performance for the Jacobi solver using matrix audikw_1 on different machines.



(a) Results from a Xeon E3123 4 core machine.

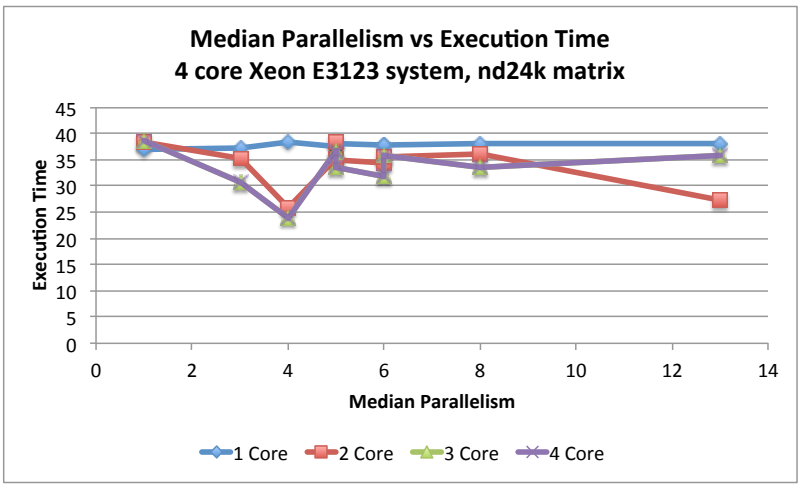


(b) Results from a Xeon E5450 8 core machine.

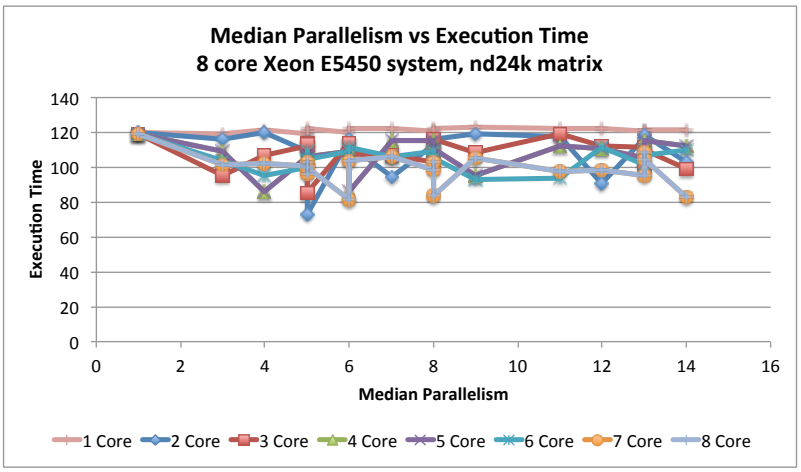


(c) Results from a Cray 24 core node

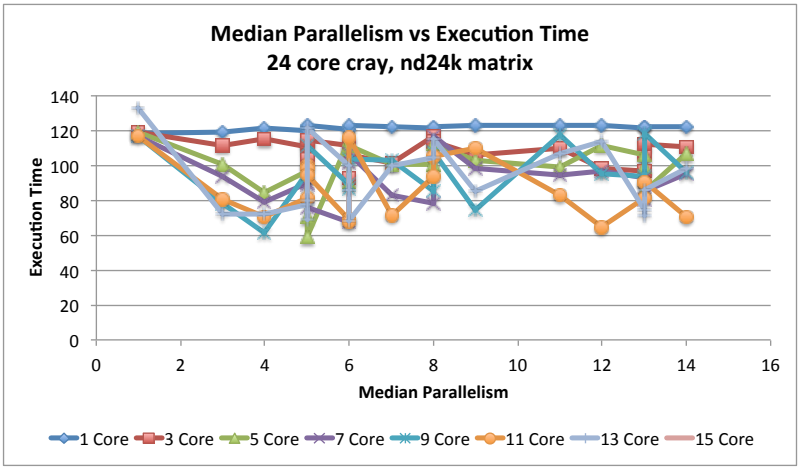
Figure 5.6: Median parallelism yielding the best performance for the Jacobi solver using matrix pwtk on different machines.



(a) Results from a Xeon E3123 4 core machine.

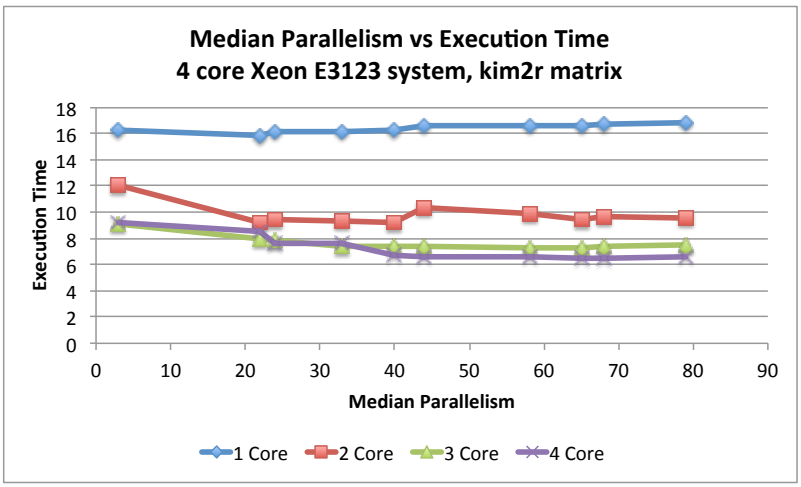


(b) Results from a Xeon E5450 8 core machine.

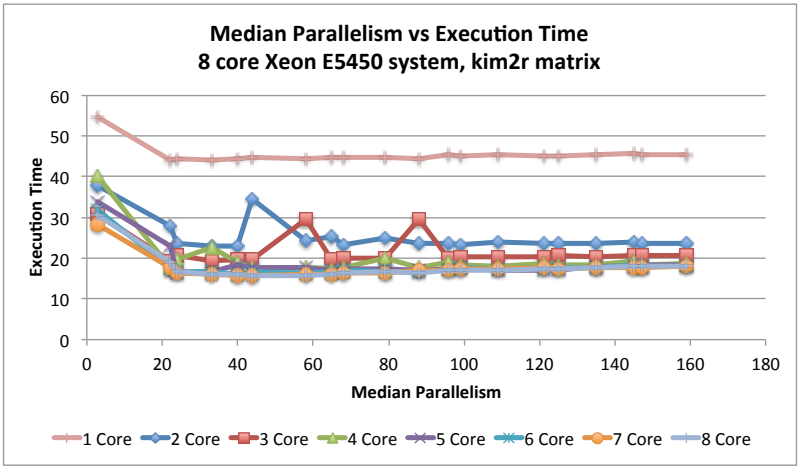


(c) Results from a Cray 24 core node

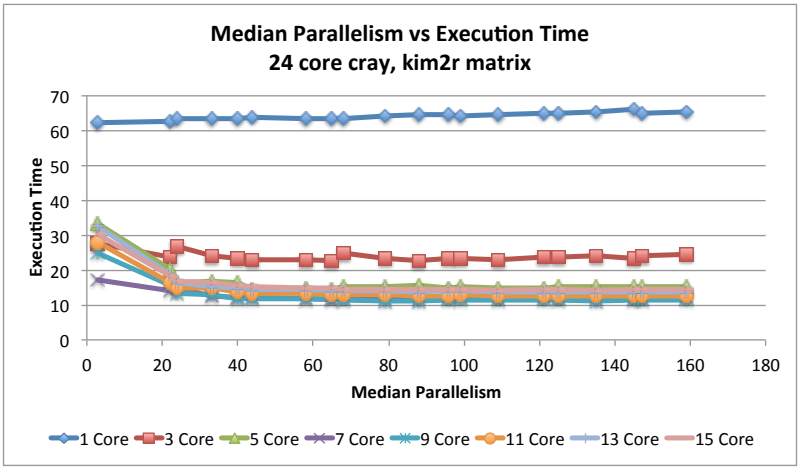
Figure 5.7: Median parallelism yielding the best performance for the Jacobi solver using matrix nd24k on different machines.



(a) Results from a Xeon E3123 4 core machine.

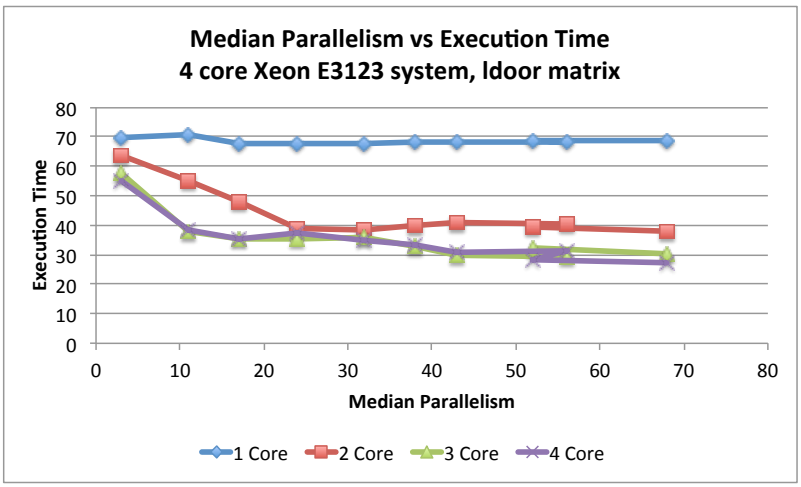


(b) Results from a Xeon E5450 8 core machine.

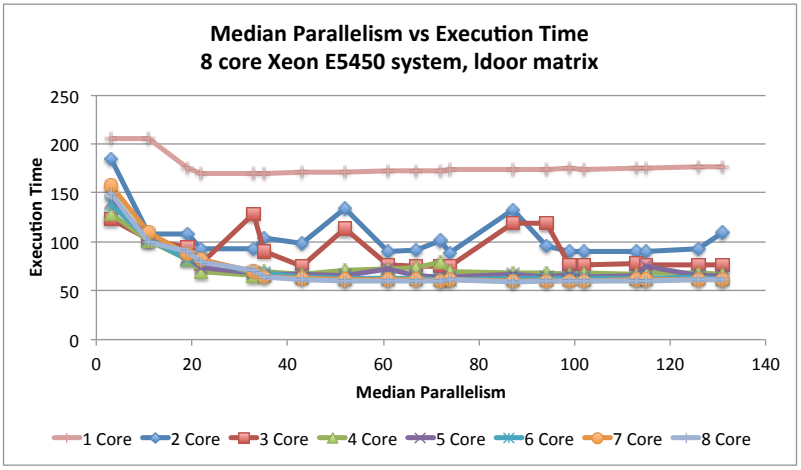


(c) Results from a Cray 24 core node

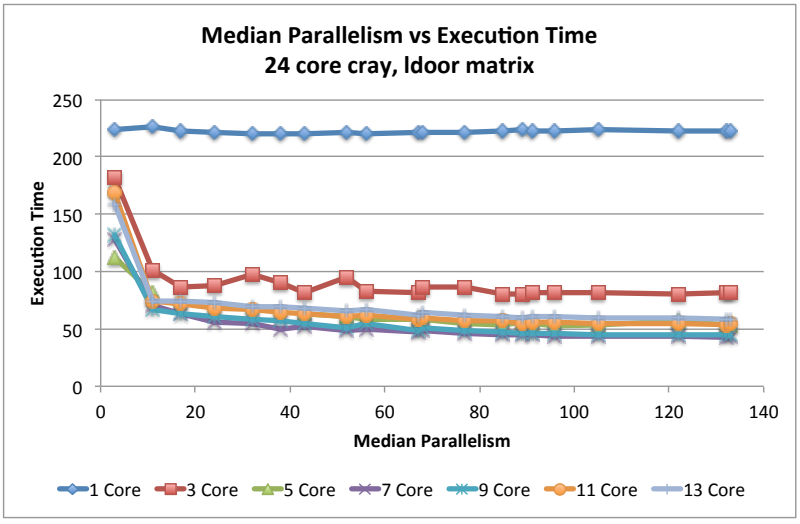
Figure 5.8: Median parallelism yielding the best performance for the Jacobi solver using matrix kim2r on different machines.



(a) Results from a Xeon E3123 4 core machine.

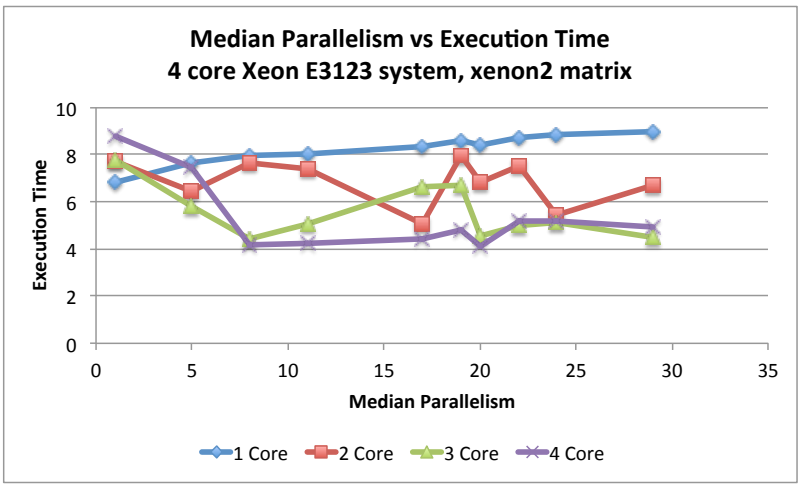


(b) Results from a Xeon E5450 8 core machine.

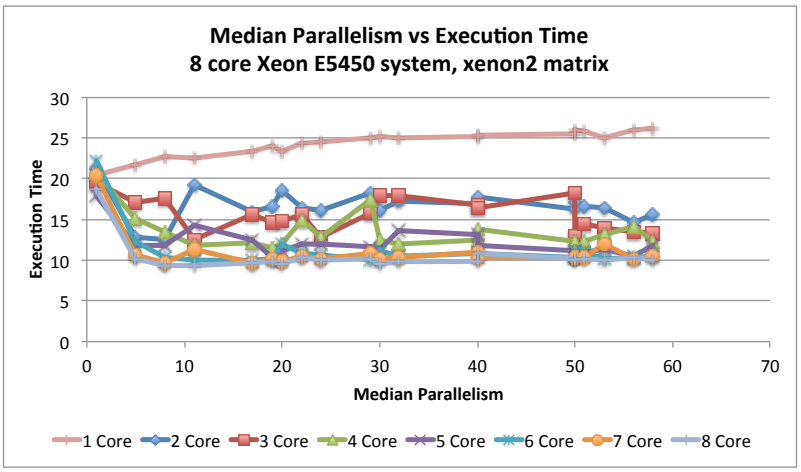


(c) Results from a Cray 24 core node

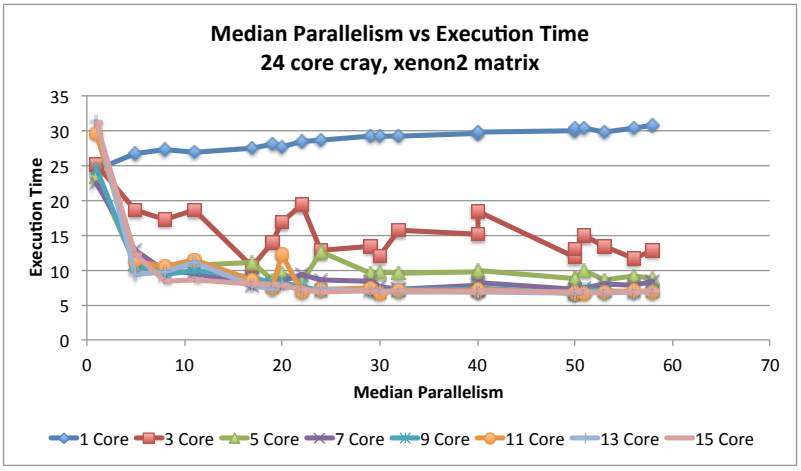
Figure 5.9: Median parallelism yielding the best performance for the Jacobi solver using matrix ldoor on different machines.



(a) Results from a Xeon E3123 4 core machine.

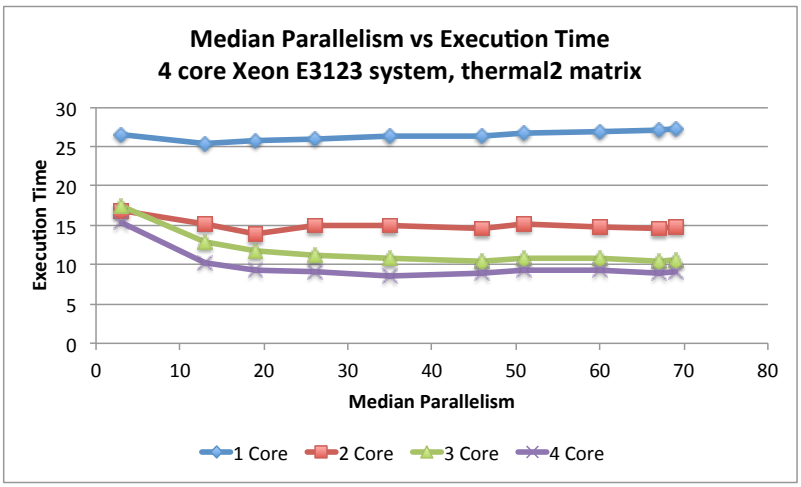


(b) Results from a Xeon E5450 8 core machine.

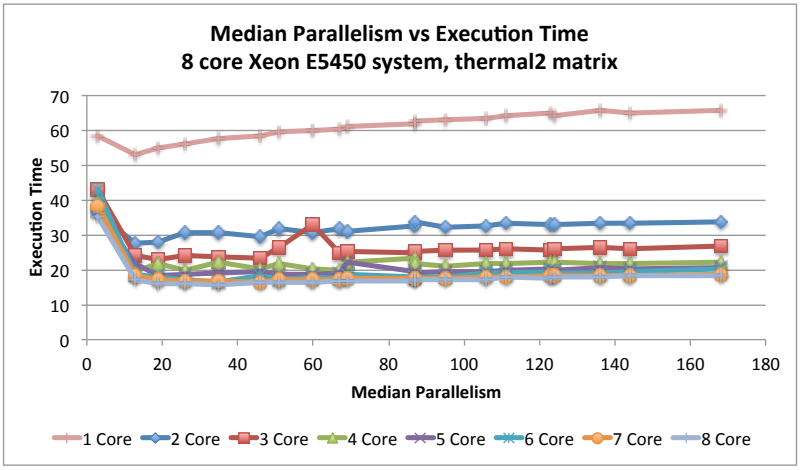


(c) Results from a Cray 24 core node

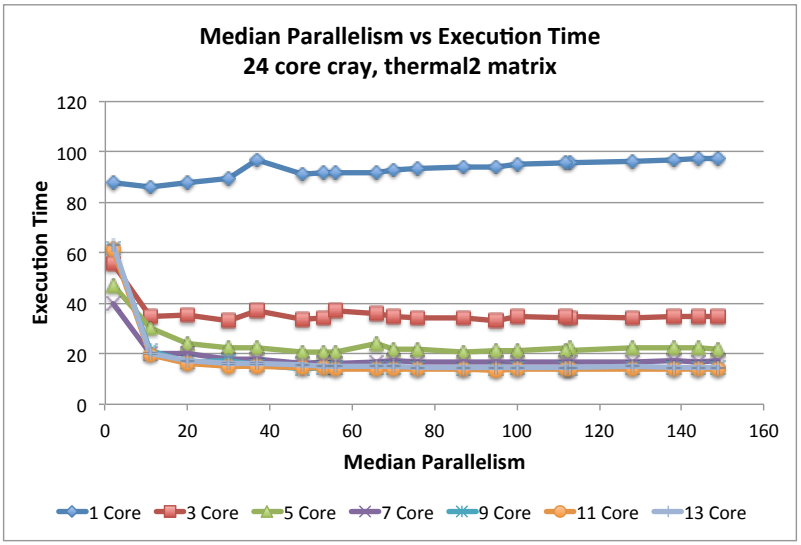
Figure 5.10: Median parallelism yielding the best performance for the Jacobi solver using matrix xenon2 on different machines.



(a) Results from a Xeon E3123 4 core machine.



(b) Results from a Xeon E5450 8 core machine.



(c) Results from a Cray 24 core node

Figure 5.11: Median parallelism yielding the best performance for the Jacobi solver using matrix thermal2 on different machines.

Chapter 6

Competing Forces In Optimization of Generalized Full Sparse Tiling

Achieving optimal performance of code using the generalized full sparse tiling algorithm involves careful tradeoffs between competing forces. In this chapter, we identify five different forces that influence the optimal tile count for a given loop chain and generalized full sparse tiling. These forces are depicted in Figure 6.1. Two forces, overhead and locality dilution, exert a force encouraging the use of fewer tiles. These are discussed in Sections 6.1 and 6.2. The other three forces, irregular tile footprint, load balancing, and parallelism, promote a higher tile count and are discussed in Sections 6.3 and 6.4. A discussion of the interaction of these forces is presented in Section 6.5.

6.1 Impact of Scheduling Overhead

One force that influences the optimal choice of tile count is the overhead of scheduling and launching each task. As the number of tasks in the task graph increases, the amount of time spent determining if a task is ready, assigning the task to a thread, setting up the context for the execution of the new task, and commencing the execution of the task increases. If the number of tasks is small and the length of each task is sufficiently long, this overhead is a small part of the overall execution time. However, as the number of tasks increases, the amount of work per task decreases. These effects combine to increase the cost of overhead time relative to useful work time.

To characterize this effect, we executed a task graph in which the task was a simple test kernel. The kernel does basic floating point computation in a loop. It makes no memory accesses. By varying the number of iterations of this computation loop, we are able to control the amount of time spent in a task. Each task was designed to take the same amount of time.

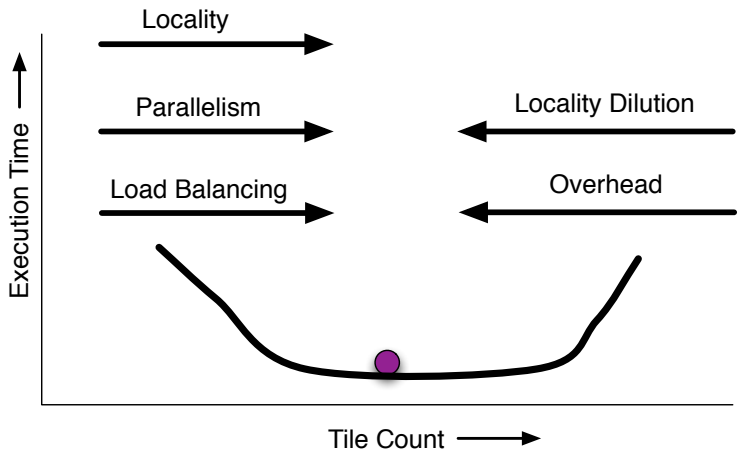


Figure 6.1: A conceptual diagram showing the different forces acting on optimal tile count.

Because different runtime systems may have different amounts of overhead, we conducted this experiment using the five different parallel execution models presented in [40].

Figure 6.2 shows the results of executing the task graph. The length of each task is decreased from 256μs to 1 μs and the percentage of unproductive, non-computation time is plotted on the vertical axis. We calculate the unproductive time by first directly measuring the total execution time using calls to a high precision hardware counter. We then multiply this time by the number of threads used. This yields the total amount of processor seconds that were available during the computation. Next, we measure the time spent in the computation portion of each task and sum that across all tasks to compute the total time actually spent doing test kernel work. The unproductive time is the difference between these two values.

At the right side of each graph there is essentially no work being done, so the non-useful work approaches 100%. As the test work length is increased, the graph execution overhead is amortized over an increasingly lengthy total execution time. Therefore, the non-useful work becomes an increasingly smaller percentage until it reaches a floor. The floor is due to some minimal overhead and load imbalance. The load imbalance increases proportionally with the test task size and thus contributes a roughly constant percentage of total execution time.

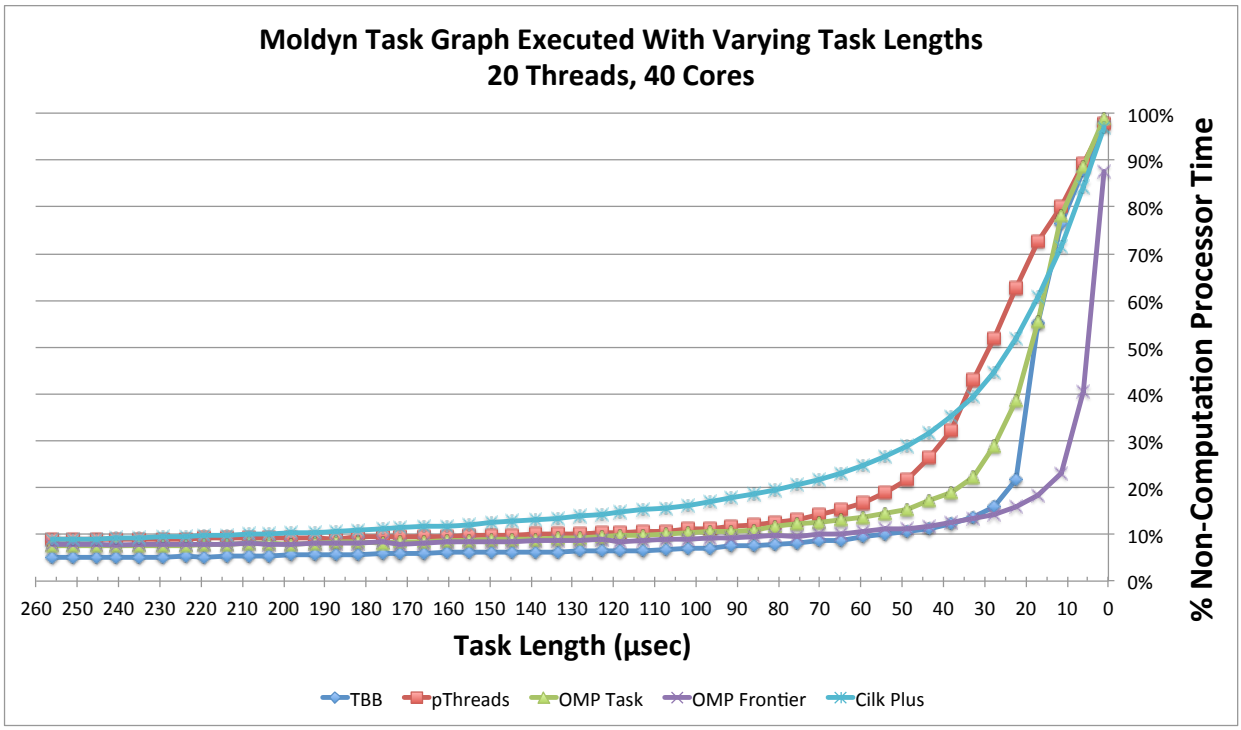


Figure 6.2: Processor time spent on unproductive work while executing a task graph. The task is simple math operations that do not access data from memory. The number of operations is varied to change the task length.

Figure 6.2 shows the graph when the experiment is run using only 20 threads on a 40 core system. The overhead for Cilk Plus takes longer to amortize, while the other engines behave similarly to one another. When the test task length reaches between 10 and 30 μs, the overhead has reached its minimum value for all engines except Cilk Plus, which requires a test task length of almost 150 μs before converging on its minimum. Notice that the floor of the graph is approximately 10%. This indicates that load imbalance contributes significantly to the overall runtime. We can also observe that if tasks execute in less than 50μs, overhead costs dominate the total runtime. While the exact percentages will vary with task graph and hardware platform, this study substantiates that overhead can be a significant force on performance and must be considered when optimizing tile count or size.

6.2 Impact of Locality Dilution

In Section 4.1, we introduced the concept of locality dilution. Locality dilution occurs when, due to data dependences, an iteration that accesses a data element is forced into a different tile than other iterations also accessing that element. This decreases the locality of a tiling and forces the processor to access data in main memory more frequently, decreasing performance.

To better visualize the effect of locality dilution, we measured the number of unique data elements accessed by all tiles in a full sparse tiling. Multiple accesses to the same element from the same tile are only counted once. Accesses to the same element from different tiles are counted once per accessing tile. Under this accounting scheme, a tiling with a single tile would have the minimum number of unique data accesses and would have no locality dilution. As the tile count increases, the probability of locality dilution increases.

Figure 6.3 shows the number of unique memory elements in a tiling of the Jacobi solver on the thermal2 sparse matrix as the tile count is increased from 1 to 200, normalized to the value found for one tile. The optimal performance of thermal2 occurs when 110 tiles are used, near the middle of the chosen range of tile counts.

If all accesses to a specific data element were confined to a single tile under each tiling, which would minimize locality dilution, then this line would remain flat at 1.0. However, we observe that the number of unique elements accessed increases as more tiles are added, indicating that locality dilution becomes increasingly worse as tile count increases. At 200 tiles, the number of unique elements accessed has increased to 1.13 times that seen when a single tile was used, indicating that elements are being accessed from multiple tiles.

Because the number of memory accesses is monotonically increasing with tile count, the number of memory accesses can be reduced simply by decreasing the number of tiles used. This finding argues in favor of minimizing the number of tiles used in a full sparse tiled loop chain.

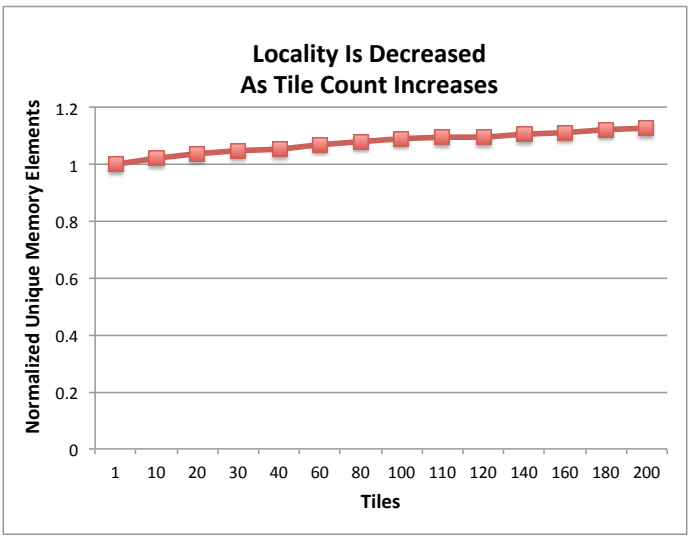


Figure 6.3: Unique memory elements accessed by a tiling as a function of tile count. This graph is from a generalized full sparse tiling of the Jacobi solver on the thermal2 sparse matrix. As the number of tiles increases, locality dilution increases, increasing the total number of memory accesses for the tiling as a whole.

6.3 Impact of Tile Irregular Data Footprint

Perhaps the most significant force driving tile count is locality. Satisfying the requirements of locality takes the form of sizing the tile irregular data footprint to fit in the appropriate level of cache. Determining the appropriate level of cache, or combination of caches, is not a simple exercise.

Most recent microprocessor designs include multiple levels of cache memory. For example, the Xeon E3-1230 microprocessor used for many of the experiments in this dissertation includes three levels of cache. The first level data cache is 32 kilobytes in size. The mid-level cache is 256 kilobytes in size and holds both data and program instructions. Both of these caches are present in each of the four cores found on the processor. A third, final level of cache is shared by all four cores. It is 8 megabytes in size and holds both data and instructions.

We focused on the mid-level cache as the target of our tile sizing experiments. This choice is driven by the results shown in Figure 6.4. Figure 6.4a shows the normalized execution time of the Jacobi solver plotted against the footprint of the irregularly accessed data in a

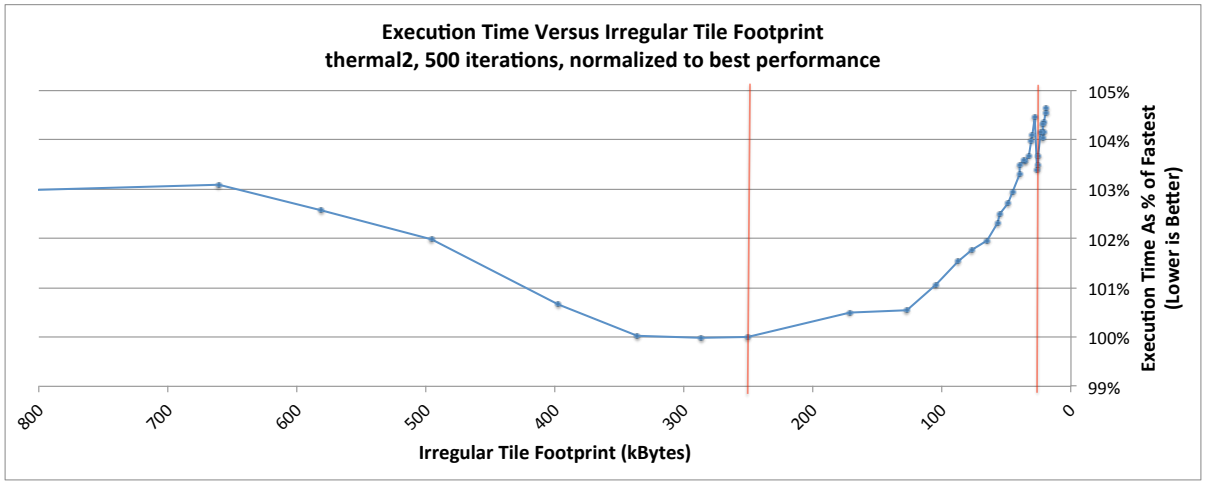
tile. As the tile footprint increases from zero, the execution time decreases. This continues until the 32 kilobyte size of the first level data cache is reached. As the tile's irregular data exceeds the size of the first level data cache, execution time jumps up. It then begins to decrease again until the irregular footprint approaches the 256 kilobyte size of the mid level cache. As the effective size of the mid level cache is exceeded, execution time increases and continues to increase as the footprint increases.

Further confirmation of the importance of the mid-level cache is shown in Figure 6.4b. This figure shows the number of mid-level cache misses plotted against the irregular tile footprint. The misses are measured using the hardware performance monitoring unit and the PAPI_L3_DCR event from the PAPI performance tool [12]. This event measures data reads that miss in the mid level cache and are then passed on to the last level cache for servicing. There is a remarkable correspondence between the performance of the loop chain and the number of mid-level cache misses. As cache misses, and therefore data requests serviced by slower memory, decrease, the execution time of the loop chain decreases.

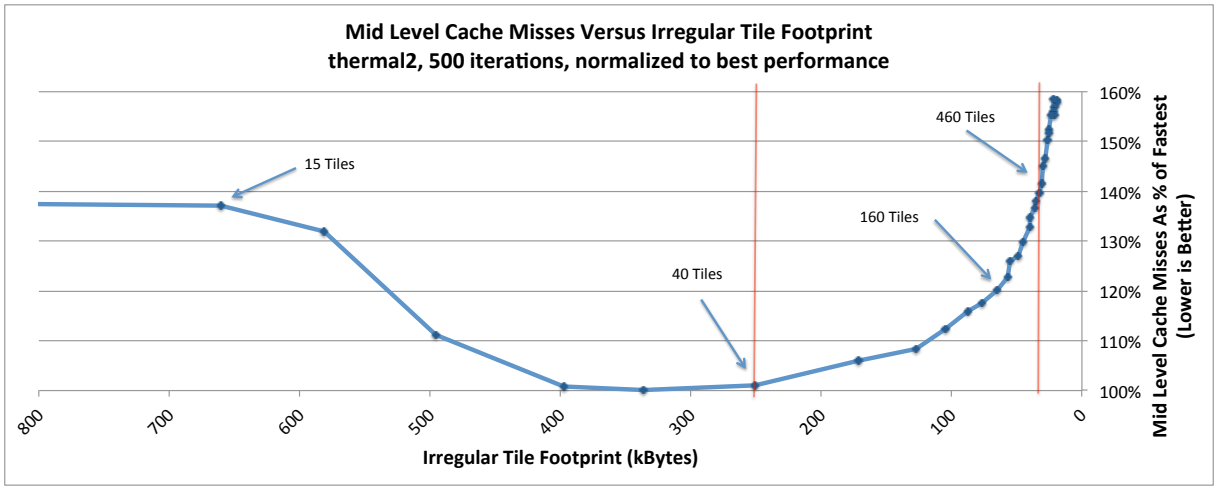
Figure 6.4 indicates that the best performance is reached when the footprint of irregular data in each tile fits into the mid level cache. While there is a performance benefit to fitting into the first level cache size of 32 kilobytes, doing so requires a high tile count. This brings with it increased overhead due to higher locality dilution, increased scheduling overhead, and time lost to the runtime system as it launches a larger number of tiles. This overhead slows down execution more than fitting into the first level cache benefits performance, so the net effect of many tiles that fit into the first level data cache is worse performance than if fewer tiles are used but irregular data footprints still fit within the mid level cache.

6.4 Impact of Parallelism and Load Imbalance

The needs of parallelism have to be considered when optimizing parallel execution time. Parallelism requirements were discussed in Section 5.3. The conclusion drawn there is that median parallelism should equal or exceed the number of cores used to execute the task

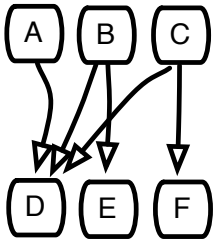


(a) Execution time versus irregular tile size.

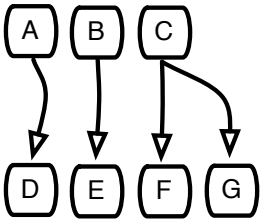


(b) Mid level cache misses versus irregular tile size.

Figure 6.4: Execution time and mid level cache misses versus irregularly accessed bytes per tile. Values have been normalized against those yielding the best execution time. Cache sizes are indicated by red vertical lines.



(a) Imbalance due to work variance within tiles.



(b) Load imbalance due to modulo effects.

Figure 6.5: Different full sparse tilings yield different amounts of parallelism and locality.

graph. If this basic requirement is not met, cores will sit idle and optimal performance will not be reached.

Load imbalance occurs when some processing elements are assigned more work than other processing elements. This results in increased execution time if some processors are idle that could be doing useful work. Load imbalance is often seen in parallel programming models in which some work is performed followed by a synchronization barrier. No processor can continue working until all processors reach the barrier, creating inefficiencies if some processors take much longer than others to reach the barrier.

In the context of generalized full sparse tiling, load imbalance occurs when the number of tasks able to execute is less than the number of cores. This can be due to insufficient parallelism in the task graph and was discussed in Section 5.3. It can also be caused by cases where there is sufficient median parallelism, but not all tiles in a level set can execute. Consider the cases in Figure 6.5. In both Figures 6.5a and 6.5b, the median parallelism is three. In Figure 6.5a, tiles A, B, and C can execute in parallel. Tiles E and F can begin as soon as tiles B and C finish, respectively. However, tile D cannot begin execution until tiles

A, B, and C have completed. If tile A completes before either tile B or C, then a processor will sit idle.

This type of imbalance can be avoided if tasks execute in nearly the same amount of time. Table 6.1 shows the average coefficient of variation for the irregular tile footprints found in the test matrices used in this work. The coefficient of variation used here is computed as the standard deviation divided by the median. This value is then averaged across tilings with tile counts from 10 to 960.

From the table, we see that there is significant variation in the amount of data accessed between tiles. For xenon2 and audikw_1, tile sizes vary by more than 20% and for nd24k they vary by nearly two thirds. Because the amount of work is proportional to the amount of data accessed, this variation tracks with variation in work and therefore in tile execution time. If the task graph contains topologies such as those depicted in Figure 6.5a, efficiency will be lost due to load imbalance.

Now consider the task graph shown in Figure 6.5b. Assume that all tasks require exactly one unit of time to complete and that three processors are being used. If this task graph executes on a machine with three cores, it will take three time units to complete. During time step one, A, B, and C will execute. During time step two, D, E, and F will execute, for example. Then, tile G will execute by itself during time step three and two processors will sit idle.

The task graph execution engines will execute the task graph in an asynchronous manner, launching tasks as soon as their predecessors are complete. This helps reduce some of the imbalance effects, but cannot completely resolve the two cases given here. One way to mitigate these issues is through the use of additional tiles. As the tile size decreases, the absolute amount of variance also decreases. Smaller tiles also reduce the cost of modulo effects. In addition, as the task graph has a higher median parallelism, there are more tasks available at any given moment, providing the scheduler with more work per core. All of these effects cause load balance and parallelism to improve with more tiles and exert a force for increasing tile count.

Table 6.1: Average coefficient of variation of irregular tile footprints for different sparse matrices. Average is taken over coefficient of variation as tile counts are swept from 10 to 960 tiles.

Matrix	Average Coefficient of Variation
thermal2	5.10%
pwtk	11.30%
kim2r	13.72%
ldoor	17.41%
xenon2	22.60%
audikw_1	27.65%
nd24k	64.46%

6.5 Interaction of Forces

We have identified five different factors that influence the optimal tile count. Specifically, these forces are locality dilution, overhead, irregular tile footprint, load balancing and parallelism. Each of these forces either drives the tile count higher or lower within certain bounds. The interaction between these forces is shown conceptually in Figure 6.1.

Two of the forces, locality dilution and irregular tile footprint, work in opposition to one another and are directly concerned with tile footprint. An effort to reduce locality dilution would increase the tile footprint. This is in opposition to efforts to reduce the tile footprint so that it fits within a cache. The other three forces, overhead, parallelism, and load balance, consider the tile count directly. As the tile count increases, so does the amount of work spent on overhead. This drives the optimal tile count down. Parallelism and load balance, on the other hand, drive the tile count up until enough parallelism is available to occupy the cores in a multicore system.

Locality dilution and overhead are minimized by setting the tile count to one. Tile size relative to cache size and the ideal amount of parallelism do not have such a well defined limit. In Section 6.3, we discussed the complexity in choosing the irregular data footprint that provides the best performance. No clear formula has emerged that universally identifies

the best tile size relative to the cache system on a given processor. However, as explored in Section 5.3, it is fairly clear how much parallelism will lead to the best performance on a multicore system.

The two key issues that remain in setting the tile count stem from knowing how many tiles will result in tiles with an irregular data footprint distribution around a set size and knowing how many tiles will produce a task graph with enough parallelism. Because of these complexities, ultimately an autotuning approach is necessary to find the optimal tile count. However, the understanding developed in this work provides some heuristics for these tuning systems. In Section 4.1.3, we suggest that aligning the 75th percentile point of the irregular data footprint distribution with the cache size is a good starting point. We also suggest a median parallelism of approximately 30 to 40. Given these recommendations, an autotuning system can select a tile count and determine the median parallelism and tile footprint. Based on the forces discussed in this section, it can then decide to increase or decrease the tile count to reach the objective.

Chapter 7

The Generalized Reordering Optimizer for Ubiquitous Tiling (GROUT) Library and Programming Interface

The Generalized Reordering Optimizer for Ubiquitous Tiling (GROUT) is a code library consisting of an extensible set of C++ classes. These classes implement the loop chain abstraction presented in Chapter 2 and provide basic inspectors and executors. GROUT also includes a complete implementation of the general full sparse tiling algorithm described in Chapters 3 through 6.

In the remainder of this chapter, we present in detail the GROUT user interface and demonstrate its use in full sparse tiling the Jacobi solver explained in Section 1.3.

7.1 Specifying the Elements of a Loop Chain

Using the GROUT programming interface, a programmer specifies the elements of a loop chain present in his or her original code using objects, such as loops, data spaces, and data access relations, that represent the parts of the loop chain abstraction. Typically, application programmers only need to construct these objects and combine them into a complete loop chain. A simple interface exists for this purpose. The programmer then passes the loop chain to inspector and executor code for processing and execution. As inspector code and runtimes may need to query loop chain elements, for example, by visiting each element in a data or iteration space, the GROUT programming interface includes additional accessor methods for use within inspectors and executors.

```
1  class IterSpace
2  {
3    public:
4      IterSpace() { }
5      virtual ~IterSpace() { }
6
7      // direct interface
8      virtual int iteration(int) const = 0;
9      virtual int size() const = 0;
10
11     // iterator interface
12     virtual IterationSpaceIterator begin() const =0;
13     virtual IterationSpaceIterator end() const =0;
14
15     ...
16 };
17
18 class ContiguousIterSpace : public IterSpace
19 {
20   public:
21     ContiguousIterSpace(int start, int inclusive_end);
22     ...
23 };
24
25 class ExplicitIterSpace : public IterSpace
26 {
27   public:
28     ExplicitIterSpace(const std::vector<int>& iters);
29     ExplicitIterSpace(const int* iters, int length);
30     ...
31 };
```

Figure 7.1: The GROUT Iteration Space Interface

7.1.1 Iteration Spaces

The `IterSpace` class in GROUT, shown in Figure 7.1, represents a loop chain iteration space. In addition to the abstract base class, GROUT supports two concrete types of iteration spaces. A programmer can use either type to represent the iteration values of loops in a loop chain. The first, `ContiguousIterSpace`, represents a one dimensional, integral tuple space with contiguous values between an inclusive lower and upper bound. This represents the iteration space of many simple `for` loops commonly found in code. The other type of iteration space is `ExplicitIterSpace`, which represents a one dimensional, integral tuple space of potentially discontiguous values such as $\{[0], [1], [44], [150]\}$. The constructor for explicit iteration spaces takes a vector or array of integers containing each of the iteration values.

When analyzing a loop, an inspector may need the iteration values within an iteration space. To facilitate this, each iteration space can be accessed in two different ways. First, the *nth* iteration tuple in an iteration space is returned by a call to the `iteration(n)` method. This method enables direct individual queries into the iteration space. This approach is facilitated by the `size()` call that returns the number of iterations in the iteration space.

The second way to access the iteration space is through `IterationSpaceIterator` objects such as are returned by the `begin()` and `end()` methods. It is important to distinguish between two definitions of the term "iteration" under consideration here. The `IterationSpaceIterator` provides a means for iterating over each index tuple of a loop's iteration space.

7.1.2 Data Spaces

In GROUT, data spaces are represented by the `DataSpace` class. This simple class can adequately express only array data spaces that have a one dimensional integral index space. This representation of data spaces covers only a small portion of the theoretical data spaces expressible by the loop chain abstraction, which, for example, includes multidimensional

```
1  class DataSpace
2  {
3    public:
4      DataSpace();
5      DataSpace(int numElements, int sizeElement=8,
6                const std::string& name = std::string("Data Space"));
7      virtual ~DataSpace();
8
9      int size() const;
10     const std::string& getName() const;
11     int getElementSize() const;
12 };
```

Figure 7.2: The GROUT Data Space Interface

arrays and associative arrays. However, in practice, this class has proven sufficient to implement all loop chains found in scientific codes we have encountered so far in this research.

A `DataSpace` has a simple constructor for use by application programmers. The constructor requires a name, an element size expressed in bytes, and the number of elements in the data space. Inspectors can retrieve this information from a `DataSpace` with calls to the `getName()`, `getElementSize`, or `size` methods, respectively. The name is primarily used for reporting and debugging purposes and would typically be set to the name of the represented array. The size of an element and the number of elements can be used by inspectors and runtimes to determine data bandwidth or footprint.

7.1.3 Data Access Relations

In the loop chain abstraction, data access relations are mappings between iteration spaces and data spaces. The GROUT library includes a base class, `AccessRelation`, that provides the basic functionality of an access relation. It can represent read, write, or combination read and write accesses used for reductions. The number of data elements in the data space that are accessed by an iteration, i, is returned by a call to the `numAccesses(i)` method. Knowing the number of accessed elements, code can make repeated calls to the `relation(i,d)` method to obtain the dth element accessed by the ith iteration. In addition, an iterator interface is also provided. The iterator iterates over each data element accessed by a specific iteration. For example, `accessRelation.begin(i)` is an iterator pointing to

```
1  class AccessRelation
2  {
3    public:
4
5      enum class AccessType
6      {
7          READ, WRITE, READWRITE
8      };
9
10     AccessRelation(IterSpace& iterspace,
11                                DataSpace& dataspace,
12                                AccessType type=AccessType::READ);
13     virtual ~AccessRelation();
14
15     // iterator interface
16     virtual AccessRelationIterator begin(int isi) const = 0;
17     virtual AccessRelationIterator end(int isi) const = 0;
18
19     // return the number of accesses from a given iteration to a given data space
20     virtual int numAccesses(int i) const = 0;
21
22     // is this a read, a write, or readwrite
23     AccessType getType() const;
24
25     // return the dsi'th relation for iteration isi
26     virtual int relation(int isi, int dsi) const = 0;
27
28     // accessors
29     const DataSpace& getDataSpace() const;
30     const IterSpace& getIterSpace() const;
31
32 };
33
34 class ExplicitAccessRelation : public AccessRelation
35 {
36   public:
37     // the map goes from iteration to data item index
38     typedef std::vector< std::vector<int> > RelationMap;
39
40     ExplicitAccessRelation(IterSpace& iterspace,
41                            DataSpace& dataspace,
42                            const RelationMap& relation,
43                            AccessType type=AccessType::READ);
44 ...
45 };
46
47 class CSRAccessRelation : public AccessRelation
48 {
49   public:
50     CSRAccessRelation(IterSpace& iterspace,
51                       DataSpace& dataspace,
52                       SparseMatrix* matrix,
53                       AccessType type=AccessType::READ);
54 ...
55 };
56
57 class IdentityAccessRelation : public AccessRelation
58 {
59   public:
60     IdentityAccessRelation(IterSpace& iterspace, DataSpace& dataspace,
61                            AccessType type=AccessType::READ);
62 ...
63 };
```

Figure 7.3: The GROUT Access Relation Interface

the first data element accessed by the *ith* iteration, while `accessRelation.end(i)` points to one element beyond the last element accessed by iteration i.

Within the GROUT library are three specific subclasses of the basic AccessRelation class. The first, `ExplicitAccessRelation`, is used to represent arbitrary mappings between iteration spaces and data spaces. At the time of construction, a data structure is passed that contains a list of data element indices accessed by each iteration. The elements in the list can be in any order and the lists are not required to have the same arity between different iterations.

The second specific access relation is the `CSRAccessRelation`. This is a specialized access relation wrapper over a sparse matrix stored in Compressed Sparse Row (CSR) format. The data elements accessed by an iteration are identified by the presence of non-zeros in the sparse matrix. This access relation class can greatly lighten the programmer's burden when creating GROUT loop chains for sparse linear algebra applications. It is also useful in other sparse applications because a CSR like structure is frequently used to store adjacency lists. Uses of a CSR index structure include representing the vertices of a mesh, storing an interactivity list for atoms in a molecular dynamics simulation, or specifying neighboring linked cells in an N-body simulation.

The final class of access relation in GROUT is the `IdentityAccessRelation`. This specialization allows a programmer to specify an identity relation between iterations and data elements. Specifically, this relation is $[i] \rightarrow [i]$ for all iterations i in the iteration space and all elements i in the identically sized data space. Identity relations occur frequently in the scientific codes we have examined. Having a specific class for this relation is a convenience for programmers and also provides a more efficient implementation than could be achieved by storing an identity relation in an ExplicitAccessRelation.

7.1.4 Loop Bodies

For the programmer, perhaps the most intrusive part of converting code to use the loop chain abstraction and the GROUT library is creating the loop body functions. These body

```
for (int i=0; i<numrows; i++) {
  double diag = 1.0;
  (*mU)[i] = 0.0;
  // loop through nonzeros for row i
  int startIndex = IA[i];
  int endIndex = IA[i+1];

  for (int p=startIndex; p<endIndex; p++) {
    int j = JA[p]; // get column index
    if (j==i) {
      diag = A[p];
    }
    else {
      (*mU)[i] += A[p] * (*mUprime)[j];
    }
  }
  (*mU)[i] = ((*mF)[i] - (*mU)[i]) / diag;
} // end looping through unknowns, second loop
```

Figure 7.4: Loop body from the Jacobi solver as found in the original code.

```
void
Jacobi::updateUOddBody(const vector<int>& iters)
{
    ScheduleIterator begin = ScheduleIterator(iters, iters.begin());
    ScheduleIterator end = ScheduleIterator(iters, iters.end());
    for ( ScheduleIterator s = begin; s != end; ++s)
    {
      int i=*s;
      double diag = 1.0;
      (*mUOdd)[i] = 0.0;

      // loop through nonzeros for row i
      // i = row, j = col, p = ptr in CSR structure
      int startIndex = IA[i];
      int startIndexOfNext = IA[i+1];

      for (int p=startIndex; p < startIndexOfNext; p++) {
        int j = JA[p]; // get column index
        if (j==i) {
          diag = A[p];
        }
        else {
          (*mUOdd)[i] += A[p] * (*mUEven)[j];
        }
      } // end for loop over non zeros

      (*mUOdd)[i] = ( (*mF)[i] - (*mUOdd)[i] ) / diag;
    } // end handling one row from one unrolling
  } // end for loop over iterations in schedule
} // end body function
```

Figure 7.5: Loop body from the Jacobi solver modified for use with GROUT. Modifications are shown in red.

functions encapsulate the original loops as found in the loop chained code, but with two modifications. The first is that the loop body code must be wrapped in a C++ function that accepts a partial schedule as its argument. This function contains a few lines that expand the passed compressed partial schedule. Each entry in the expanded schedule is used as the loop index for the one original loop contained in the function. That loop otherwise matches the original loop body.

Figure 7.4 shows the original code for one of the loops in the Jacobi solver. Figure 7.5 shows the same loop after it has been modified for use with GROUT. Note that line 1 in Figure 7.4 has been expanded to lines 1-8 in Figure 7.5 to enable non-sequential iteration over the loop.

GROUT takes advantage of the `std::function` class introduced in the C++11 standard to define the loop body functions. Any value that can be assigned to an object of type `std::function<void(const std::vector<int>&)>` can be used as a loop body. This includes functions, class methods, lambda functions, and functors. This offers the programmer significant flexibility in defining loop bodies. For example, unlike typical function pointers, a class method defined within an object can be used. This allows a programmer to encapsulate the data needed by a loop chain within an object and code loop bodies as member functions in that class. Alternatively, the loop body can be directly specified in the code using a lambda function such as `std::function<void(const std::vector<int>&)> body = [](const std::vector<int>& iters) { ... }` . Because lambda functions act as closures, they can capture the value of variables in the defining context and later make those values available to the loop body.

The use of the `std::function` class for loop bodies also has the advantage that loop bodies can be dynamically replaced between executions of a loop chain. This allows a runtime system or just-in-time compiler to dynamically optimize the loop body and replace it with more efficient code.

```
class Loop
{
  public:
    Loop(std::function<void(const std::vector<int>&)> body, const IterSpace& iters);
    virtual ~Loop();

    // access relations
    int addAccessRelation(const AccessRelation* rel);
    int getNumAccessRelations() const;
    const AccessRelation* getAccessRelation(int relNum) const;

    // loop body
    std::function<void(const std::vector<int>&)> getLoopBody() const;

    // iteration space
    const IterSpace& getIterSpace() const;

};

class LoopChain
{
  public:
    LoopChain();
    virtual ~LoopChain();

    int addLoop(const Loop& newloop);
    int getNumLoops() const;
    const Loop& getLoop(int loop) const;

};
```

Figure 7.6: The GROUT Loop and LoopChain Interface

7.1.5 Loops and Loop Chains

Once all of the pieces of a loop chain have been instantiated using GROUT, they must be assembled into individual loop abstractions and finally into a complete loop chain. Loops are represented in GROUT by objects of the class `Loop`. The constructor requires a loop body and an iteration space. Once a `Loop` object is instantiated, AccessRelations can be added to it via calls to the `addAccessRelation` method. The remainder of the methods, `getAccessRelation`, `getLoopBody`, and `getIterSpace` are used to retrieve the elements of the loop abstraction.

A `LoopChain` object is used to encapsulate all the loops in a loop chain. Each loop is added to the chain using the `addLoop` method. The order loops are added is their relative order in the chain, with the first loop added being the first loop in the chain and so forth. Loops in the chain can be extracted using the `getNumLoops` and `getLoop` methods.

```
1  class Inspector
2  {
3
4    public:
5      Inspector(const LoopChain& chain);
6      virtual ~Inspector();
7
8      // execute the inspector
9      virtual void inspect()=0;
10
11     // set the
12     void setExecutionTask(const Task& task);
13
14     // get information used by executors
15     const Schedule& getSchedule() const;
16     const TaskGraph& getTaskGraph() const;
17
18 };
```

Figure 7.7: The GROUT Inspector Interface

7.2 Applying Optimizations Using Inspectors

The GROUT library supports run time optimizations that take an inspector/executor approach [8]. In this scheme, a loop chain is passed to the inspector, which examines it and performs some type of optimization such as data or iteration reordering. The GROUT library includes an abstract class, `Inspector`, that defines the interface for all inspectors. Figure 7.7 shows this interface.

An `Inspector` is used as follows. First, an inspector object is constructed from information provided in a supplied `LoopChain` object. Next, when that `LoopChain` is fully defined and the programmer is ready to have it inspected, a call to the `inspect` method is made. This call performs any necessary analysis and may populate a `Schedule` object containing a new ordering for the loop iterations in the loop chain. It may also store in a task graph structure a partial ordering on task execution. The schedule and task graph may later be used by an `Executor` that runs the loop chain following the reordered schedule.

The principal inspector included in GROUT is the `FullSparseTiler`. This implements the generalized full sparse tiling algorithm described in Chapter 3. The constructor takes in a `LoopChain` object, but also requires the number of tiles that should be used for tiling, the number of threads that the parallel inspector should use when running, the type of

```
1  class Executor : public Algorithm
2  {
3   public:
4    Executor(const LoopChain& chain, const Inspector& insp, int numThreads);
5    virtual ~Executor();
6
7    virtual int initialize();
8    virtual int execute();
9
10 };
```

Figure 7.8: The GROUT Executor Interface

partitioner that should be employed, and the loop number that should be used for seed partitioning. If the number of threads is set to negative one, the code will use one thread per virtual processor in the system. If the seed space is set to negative one, a heuristic will be used to select an appropriate seed space.

7.3 Executing Loop Chains Using Executors

`Executor` objects are used in GROUT to execute in parallel the schedules generated by `Inspector` objects. The interface is quite simple and consists of only two functions beyond the constructors and destructor, as shown in Figure 7.8. The constructor takes the loop chain and inspector objects as well as the number of threads that should be used for executing the loop chain. Note that a different number of threads can be used for execution than was used for inspection. The `Executor` queries the `Inspector` to obtain the schedule and task graph, which the executor uses to determine the execution order of iterations.

Before the loop chain is executed, the `Executor` must be initialized. This call sets up the parallel execution engine and integrates the engine with the task graph to be executed. The executor should be initialized only after the inspector has completed its inspection and has generated the final task graph. The `initialize` method need only be called once, regardless of the number of times the loop chain is executed. To execute the loop chain, the programmer calls the `execute` method one time per execution of the loop chain. No additional code or parameters are needed.

```
1  ...
2    for (int i=0; i<numrows; i++) {
3      double diag = 1.0;
4      Ueven[i] = 0.0;
5      for (int p=IA[i]; p < IA[i+1; p++) {
6        int j = JA[p];
7        if (j==i) { diag = A[p]; }
8        else { Ueven[i] += A[p] * Uodd[j]; }
9      }
10     Ueven[i] = (F[i] - Ueven[i]) / diag;
11   }
12
13 ...
```

Figure 7.9: The first inner loop of the sparse Jacobi solver kernel for CSR sparse matrices.

```
1  // Declare a loop chain
2  LoopChain jacobiChain;
3
4  // Create the iteration space
5  int numberOfRows = matrix->getNumRows();
6  ContiguousIterSpace iterAllRows(0,numberOfRows-1);
7
8  // Create the data spaces
9  DataSpace UOdd(numberOfRows);
10 DataSpace UEven(numberOfRows);
11
12 // Create the access relations
13 CSRAccessRelation relReadUOdd(iterAllRows, UOdd, matrix, AccessType::READ);
14 IdentityAccessRelation relWriteUEven(iterAllRows, UEven, AccessType::WRITE);
15
16 // Create and define the Loop object
17 Loop loopUpdateUEven(bodyJacobiUpdateUEven, iterAllRows);
18 loopUpdateUEven.addAccessRelation(&relReadUOdd);
19 loopUpdateUEven.addAccessRelation(&relWriteUEven);
20
21 // Add the Loop to the LoopChain
22 jacobiChain.addLoop(loopUpdateUEven);
```

Figure 7.10: The Jacobi solver loop from Figure 7.9 specified using the GROUT API.

7.4 A Complete Example Using GROUT

In the section, we present an example using the GROUT library on the Jacobi solver first introduced in Section 1.3. This example illustrates the entire loop chain definition and full sparse tiling processes. Each of the individual loop chain components was described in Sections 7.1, while the inspector and executor were described in Section 7.2 and Section 7.3.

Figure 7.9 contains the original Jacobi kernel code for the first of the two unrolled loops shown in Figure 1.1. Figure 7.10 shows the same code implemented using the GROUT API. Line 2 declares the overall `LoopChain` object. Next, the various parts of the loop definition

```
1
2  // Create the FullSparseTiler inspector and perform the inspection
3  FullSparseTiler fst(chain, numTiles, numThreads, partitionerType);
4  fst.inspect();
5
6  // Create the Executor
7  Executor* executor = new Executor(jacobiChain, fst, numThreads);
8  executor->initialize();
9
10 // call the loop chain executor once for each two outer loop iterations
11 for (int i=0; i < numIters; i += step)
12 {
13     executor->execute();
14 }
```

Figure 7.11: The Full Sparse Tiling Inspector and Executor for the Jacobi solver.

are created. Line 6 declares the iteration space, a `ContiguousIterSpace`, containing one iteration for each of the rows in the sparse matrix. The data spaces, `UEven` and `UOdd`, are declared on lines 9 and 10, each having one element per row in the sparse matrix. Lines 13 and 14 create the access relations using the data spaces. The first, a `CSRAccessRelation`, defines the read access relation between the iteration space and the `UOdd` data space. The second access relation, an`IdentityAccessRelation`, maps between the same iteration space and writes to the `UEven` data space. In lines 17 through 19, the `Loop` object is created and the access relations are added to it. Finally, the loop is added to the `LoopChain` in line 22.

Figure 7.11 shows the code to create and use the inspector and executor for the Jacobi example. Lines 3 and 4 create the `FullSparseTiler` inspector and call it to perform the inspection. Lines 7 and 8 create the executor and initialize it for use with the schedule and task graph produced by the inspector. Finally, lines 11 through 13 call the executor's `execute` method once for each two iterations of the original outer loop.

This example serves to illustrate the strengths and weaknesses of using a library based approach such as GROUT. A programmer with a basic knowledge of loop chains can make the required code changes and GROUT library calls. No knowledge of data dependency analysis or full sparse tiling is required. The process of generating the library calls shown here can be largely automated by a compiler. In particular, the iteration spaces and loop bodies can be automatically identified in much the same way parallel loops are discovered by

OpenMP aware compilers. Even in the case that these elements are automatically identified, the responsibility of defining data access relations will remain with the programmer.

Chapter 8

Conclusions And Future Research

In this chapter, we review the most significant conclusions reached by this dissertation effort. These findings relate both to loop chains and to generalized full sparse tiling. We also briefly outline some related future research that naturally extends this body of work.

8.1 Conclusions

A significant contribution of this research is the formal definition of a new programming abstraction, the *loop chain*. The ubiquity of the loop chain code pattern, a sequence of parallel loops with no intervening code, underscores the importance of the loop chain abstraction. Loop chains have been identified in computational fluid dynamics applications such as the OP2 airfoil benchmark and the HYDRA code used by Rolls-Royce. Loop chains are also present in molecular dynamics programs such as the CoMD molecular dynamics benchmark for materials science and the miniMD molecular dynamics benchmark from Sandia National Laboratory. Sparse linear algebra applications, such as the Jacobi solver studied in this work and the matrix powers kernel, also contain loop chains.

The loop chain abstraction provides advantages over existing approaches for both the application programmer and the run-time optimizer. Loop chains enable application programmers to express that a sequence of loops has certain properties. In Section 2.3, we present several possible methods for a programmer to express a loop chain. These different methods vary in how much of a burden they place on the programmer, but all are simple enough to be used by programmers in practice.

A key portion of the loop chain specification are data access relations. These relations stem directly from memory references in the source code. Because of this, access relations can

be more easily understood and identified by programmers than more abstract and decoupled data dependence relations between loop iterations.

Loop chains also provide a clean abstraction from the perspective of the run-time optimizer as well. Different run-time optimizers can be written to schedule iterations of loop chains without having to directly interact with source code. Likewise, these tools do not require abstract syntax trees or other compiler intermediate representations that typically are not available during program execution. All of the information needed is available from the loop chain abstraction once program data has been read into memory. This makes loop chains a good abstraction for use at inspector time.

Because inspectors can be written that rely only on information provided by the loop chain abstraction, these inspectors can be much more general than implementations that relied on a specific loop and data dependence structure. This property of loop chains facilitated the development of the generalized full sparse tiling algorithm. The generalized full sparse tiling algorithm can reschedule the iterations of any loop chain using only the information provided within the loop chain abstraction. All previous full sparse tiling implementations were written for a specific application and could not be easily applied to other applications.

The GROUT library further lowers the barrier to entry for full sparse tiling usage. This C++ library provides a reference implementation for both loop chains and generalized full sparse tiling. It also includes runtime engines to execute task graphs using a variety of shared memory parallel programming models.

We have examined the performance of generalized full sparse tiled code on a range of shared memory multicore systems. From this study, we have found that five forces impact the selection of tile size. These forces are overhead, locality dilution, locality, parallelism, and load balancing. Balancing these forces is accomplished by selecting a tile size with an irregular tile footprint that fits within the effective size of the cache, then increasing the tile count until sufficient tiles are available to satisfy the requirements of parallelism and load balancing.

8.2 Areas Identified for Further Research

This work could be extended in many different ways, but three key directions are outlined here. First, the generalized full sparse tiling algorithm could be extended to support execution of loop chains on distributed memory systems. Second, additional research could be conducted regarding how to express loop chains using domain specific languages. Finally, there are several points within the full sparse tiling process at which locality could be improved even further than it currently is.

8.2.1 Full Sparse Tiling for Distributed Memory Systems

Another future research effort would be to extend the general full sparse tiling algorithm and the task graph execution engines to distributed memory hardware. If done in a general way, this work could support accelerators such as graphic processing units (GPUs) or many integrate core (MIC) architectures such as Intel's Xeon Phi as well as traditional clusters of multicore processors.

Under one possible approach, executing a task graph on a distributed memory system would involve partitioning the task graph between nodes or memory domains. This could be done statically, using an algorithm to balance the load appropriately. Within each node, the portion of the task graph assigned to the node could be executed using the existing task graph execution engines that exploit asynchronous parallelism and dynamically load balance. This approach also naturally provides nested or hybrid parallelism, as tasks are assigned first to a node and then execute using all the processors available within the node.

Assuming a task partitioning with specific tasks assigned to each memory domain, communication and data distribution assignments could be made. For loop chain invariant data, a naïve implementation could simply duplicate all static data in all memory domains. More sophisticated implementations could duplicate only the necessary data needed per domain, thereby reducing the memory footprint and initial communication volume. Copying data to distributed memory domains introduces the opportunity to compress or reorder the data

as part of the duplication process. These schemes would then have to modify the memory references in the loop bodies to properly access the relocated data.

Communication of data modified during the course of loop chain execution would also have to be addressed. This task is simplified somewhat because the task graph contains edges between any tasks that share data. If the full sparse tiling process were modified to decorate those edges with references to specific data items, the inter-node communication pattern could be read from the task graph directly. For communication between memory domains, a communication proxy node could be added to the task graphs assigned to the sending and receiving memory domains. This task could communicate using MPI calls. On the sending side, this proxy task would be a successor to tasks producing needed data. It would wait until the results were ready, then gather the results and send them to the remote domain. On the receiving side, the receiving proxy node would receive the data, place it into the memory image of that memory domain, and then signal the task graph engine on the receiving side that the predecessor task had completed. In this way both data and synchronization could be handled with a single mechanism and the majority of the task graph execution engine would be oblivious to its running on a distributed memory system.

Open research questions in this area include how best to partition the task graph and distribute pieces over the different nodes in the distributed memory system. Work is also needed to discover the most efficient way to remap memory accesses to point to the data duplicated on each node. There are also opportunities for communication aggregation that should be explored.

8.2.2 Extending Domain Specific Languages To Define Loop Chains

There are a number of ways that a programmer can express a loop chain. These different approaches were discussed in Section 2.3. For this dissertation, the GROUT library based approach to specifying loop chains was developed. A natural continuation to that work would be to implement loop chains as part of a domain specific language. The DSL compiler

could ascertain the loop chain properties either through inspection or from loop chain specific language extensions. These properties could then be emitted as calls to the GROUT library.

Existing DSLs for the specification of computation over meshes, such as OP2 [11], Chombo [21], and Liszt [27], already require the programmer to specify computation as traversals over mesh elements. Any of these DSLs would make a good starting point for evaluating loop chain identification within the context of an established domain specific language.

8.2.3 Locality Improvements to the Generalized Full Sparse Tiling Algorithm

A few areas for potentially improving the generalized full sparse tiling algorithm have been identified. These ideas center around ways to further improve both the temporal and spatial locality seen when executing full sparse tiles.

As presently constituted, the general full sparse tiling algorithm does not tile together loop iterations based on input dependences. Only flow, anti, or output dependences are considered by the algorithm, as they impose constraints on iteration ordering. Therefore, if iterations of two different loops read from data that is not written during the loop chain, they will not be grouped together. The impact of considering input dependences during tiling remains an open research question.

The potential benefits of reordering the iterations of a loop assigned to a tile should also be explored. The task graph abstraction ensures that any ordering of loop iterations within a single loop is valid. At present, iterations of a loop assigned to a tile are executed in ascending order. However, there may be performance benefits to executing the iterations in a different, optimization order. It may improve locality if iterations within a loop were sorted by data items accessed in the iterations, as this may reduce the reuse distance between accesses to less than that of the lowest level cache. It may also improve spatial locality.

Likewise, sorting iterations by the data elements they access may also improve temporal locality between loop iterations of different loops in a tile. For example, there are relatively

fewer memory accesses between the first executed iteration of loop L_0 and the *first* executed iteration of loop L_1 than there are between the first iteration of L_0 and the *last* iteration of L_1. If the iterations of each loop were sorted by data accessed, then the first iterations of each loop would access similar data as would the last iterations of the two loops. In theory, this would reduce the reuse distance between accesses to the same data elements and may lead to a higher cache hit rate.

8.3 Summary

The loop chain programming abstraction is a new abstraction that bridges the gap between a programmer's conception of a sequence of loops and the data structures needed by a run-time inspector or optimizer. One of its key benefits is that it enables the generalization of optimizer code. In this work, this was demonstrated by the generalization of the full sparse tiling algorithm. In the course of generalizing the full sparse tiling algorithm, various supporting algorithms and codes were developed. These supporting efforts include a fast shared memory parallel graph and hypergraph partitioner and a collection of task graph execution engines built on top of a wide variety of parallel programming models. A reference implementation of all these elements, including the loop chain abstraction, the generalized full sparse tiling algorithm, the partitioners, and the task graph execution engines, is available as part of the GROUT C++ library. Using a Jacobi sparse linear equation solver optimized using the GROUT library, we conducted a series of experiments to understand the interplay between forces that impact parallel performance. These experiments led to an improved comprehension of the full sparse tiling algorithm and how it can be tuned to achieve maximum performance.

References

[1] M. F. Adams and J. Demmel. Parallel multigrid solver algorithms and implementations for 3D unstructured finite element problem. In *Proceedings of SC99: High Performance Networking and Computing*, Portland, Oregon, November 1999.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, Reading, MA, second edition, 2007.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, Feb. 2011.

[4] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[5] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. D. Vorst. Templates for the solution of linear systems: Building blocks for iterative methods, 1994.

[6] F. Bassetti, K. Davis, and D. Quinlan. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. *Lecture Notes in Computer Science*, 1505, 1998.

[7] A. Basumallik and R. Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, New York, NY, USA, 2006. ACM Press.

[8] D. Baxter, R. Mirchandaney, and J. H. Saltz. Run-time parallelization and scheduling of loops. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, SPAA '89, pages 303–312, New York, NY, USA, 1989. ACM.

[9] G. Belter, E. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 1–12, New York, NY, USA, 2009. ACM.

[10] C. Bertolli, A. Betts, N. Loriant, G. Mudalige, D. Radford, D. Ham, M. B. Giles, and P. Kelly. Compiler optimizations for industrial unstructured mesh cfd applications on gpus. In *Accepted for publication at Languages and Compilers for Parallel Computing Workshop*, 2012.

[11] C. Bertolli, A. Betts, G. Mudalige, M. Giles, and P. Kelly. Design and performance of the OP2 library for unstructured mesh applications. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7155 of *Lecture Notes in Computer Science*, pages 191–200. Springer, 2012.

[12] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors,. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[13] U. Catalyurek and C. Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, pages 118–, Washington, DC, USA, 2001. IEEE Computer Society.

[14] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Computing*, 10:673–693, 1999.

[15] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[16] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 123–132, New York, NY, USA, 2008. ACM.

[17] A. Chandramowlishwaran, K. Knobe, and R. W. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[19] L. Chen, Z.-Q. Zhang, and X.-B. Feng. Redundant computation partition on distributed-memory systems. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 252 –260, oct. 2002.

[20] P. Cicotti and S. Baden. Latency hiding and performance tuning with graph-based execution. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on*, pages 28 –37, oct. 2011.

[21] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. V. Straalen. Chombo software package for AMR applications: design document. `http://davis.lbl.gov/apdec/designdocuments/chombodesign.pdf`.

[22] J. Culberson. Graph coloring programs. http://webdocs.cs.ualberta.ca/~joe/coloring/.

[23] L. Dagum and R. Menon. OpenMP: An industry-standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[24] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1 – 1:25, 2011.

[25] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[26] P. J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, July 2005.

[27] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[28] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 229–241, New York, NY, USA, 1999. ACM.

[29] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis*, pages 21–40, February 2000.

[30] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta. Extending the openmp tasking model to allow dependent tasks. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, pages 111–122, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons Inc., New York, 2005.

[32] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.

[33] M. Forum. MPI : A message - passing interface standard. *The International Journal of Supercomputing and High Performance Computing*, 8(3-4):159–416, 1994.

[34] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007.

[35] Intel Corporation. *Intel Threading Building Blocks Reference Manual*, 2009.

[36] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 343–348, New York, NY, USA, 1999. ACM.

[37] P. H. J. Kelly, O. Beckmann, T. Field, and S. B. Baden. Themis: Component dependence metadata in adaptive parallel applications. *Parallel Processing Letters*, 11(4):455–470, 2001.

[38] C. D. Krieger, F. Luporini, C. Bertolli, G. teodor Bercea, C. Olschanowsky, M. M. Strout, and P. H. J. Kelly. Tiling loop chains in unstructured mesh applications. 2014.

[39] C. D. Krieger and M. M. Strout. Executing scientific task graphs using Concurrent Collections. In *The Third Annual Concurrent Collections Workshop (CnC)*, September 2011.

[40] C. D. Krieger and M. M. Strout. Executing optimized irregular applications using task graphs within existing parallel models. In *Proceedings of the Second Workshop on Irregular Applications: Architectures and Algorithms (IA^3) held in conjunction with SC12*, November 11, 2012.

[41] C. D. Krieger and M. M. Strout. A fast parallel graph partitioner for shared-memory inspector/executor strategies. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, September 2012.

[42] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. Kelly, G. Mudalige, B. Van Straalen, and S. Williams. Loop chaining: A programming abstraction for balancing locality and parallelism. In *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Boston, Massachusetts, USA, May 2013 (Under Submission).

[43] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM.

[44] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[45] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 140–152, 1988.

[46] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 36:1–36:12, New York, NY, USA, 2009. ACM.

[47] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[48] W. Pugh and E. Rosser. Iteration space slicing and its application to communication optimization. In *Proceedings of the 11th international conference on Supercomputing*, pages 221–228. ACM Press, 1997.

[49] W. Pugh and E. Rosser. Iteration space slicing for locality. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, volume LNCS 1863, pages 164–184, London, UK, August 1999. Springer-Verlag.

[50] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 72:1–72:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[51] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 72:1–72:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[52] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991.

[53] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, June 2003. ACM.

[54] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck. Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, College Park, Maryland, July 2002.

[55] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse tiling for stationary iterative methods. *Int. J. High Perform. Comput. Appl.*, 18(1):95–113, 2004.

[56] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications*, 18(1):95–114, February 2004.

[57] S. Wood. Smores: Sparse matrix omens of reordering success, 2010.

[58] S. Wood. Sparse matrix power kernel and matrix reorderings. In *Grace Hopper Celebration of Women in Computing*, 2010.

[59] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM.