Transforming Sparse Matrix Computations

by

Kazem Cheshmi

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

Department of Computer Science
University of Toronto

Transforming Sparse Matrix Computations

Kazem Cheshmi
Doctor of Philosophy

Department of Computer Science
University of Toronto
2021

Abstract

Sparse matrix computations are at the heart of many scientific applications and data analytics codes. The performance and memory usage of these codes depend heavily on their use of specialized sparse matrix data structures that only store the nonzero entries. However, such compaction is done using index arrays that result in indirect array accesses such as `A[B[i]]` where `A` and `B` are both arrays. Numerical libraries can provide high-performance code for an individual sparse kernel however they must be manually tuned and optimized for different inputs and architectures. Alternatively, compilers are used to optimize codes that provide architecture portability. Due to these indirect array accesses, memory access information is unknown at compile-time, and thus it is challenging to vectorize a sparse matrix method or run it in parallel cores.

To automate the generation of code for efficient execution of sparse code, several compile-time and runtime techniques are required. Existing techniques are either not efficient or need manual effort to extend to different sparse matrix computations. Consequently, in this dissertation, I address the problem of automating the optimization of sparse matrix code on parallel processors with a specific focus on sparse linear solvers and numerical optimizations.

This dissertation presents a set of code transformations and algorithms, all implemented in a novel code generator called Sympiler, that automates the optimization of sparse matrix codes on parallel processors. Sympiler takes a sparse method, arising from a sparse linear system or sparse numerical optimization, and decouples

information related to the computation pattern of the method, i.e., symbolic information, and uses this information to transform the code to vectorizable and parallel code. Sympiler also enables the reuse of symbolic information when the computation pattern remains static for a period of time in the simulations or for when it changes modestly. Evaluation results show the automatically generated code by Sympiler provides between 1.2–3.1× speedup compared to highly optimized sparse linear solver libraries. It also improves the performance of sparse numerical optimization such as Quadratic Programming between 1.7–24.8× compared to highly efficient solvers.

Acknowledgments

I would like to express my greatest gratitude to Maryam Mehri Dehnavi, my advisor and my mentor. I adore all discussions and long arguments we had for polishing the problems and the ideas throughout my PhD program. Maryam gave me the opportunity to grow and also to build what I was dreaming for. I also thank my committee members Ken Jackson, Angela Demke Brown, and Christina C. Christara for their feedback and insightful questions and also Christopher Batty for his very helpful suggestions that contributed significantly to further improving the dissertation.

I would like to especially thank my collaborators and mentors, Michelle Mills Strout, Shoaib Kamil, and Danny M. Kaufman who I learned a lot from. They have helped in numerous aspects to polish and develop the ideas in this thesis. Without Michelle and Shoaib, Sympiler would have not existed! Danny played a crucial role in making Sympiler scalable and applying it to computer graphics, helping its impact significantly.

I also want to thank my mentors from Rutgers and Concordia universities. Santosh Nagarakatte and Saman A. Zonouz provided invaluable advice on how to navigate my way around research and how to position myself for a future career in academia. Maria A. Amer and Jelena Trajkovic also helped a lot in the early stages of my PhD and so did my MSc advisors Siamak Mohammadi, Daniel Versick, and Djamshid Tavangarian.

I would like to thank my co-workers at Paramathics, Zachery Centinic, Bangtian Liu, Saeed Soori, and Behrooz Zare for all the good memories. I especially thank Zachery and Behrooz who help extend Sympiler to more applications. I thank my friends who have patiently tolerated my very busy schedule and encouraged me along the way. I especially thank Fattaneh Jafari who introduced me to Maryam where this PhD began. It is so sad we lost such a beautiful soul recently. I wish her soul to rest in peace.

Last but not the least, I am most grateful to my siblings, Mehdi, Hadi, Ali and Leila and also my parents. Their support both mentally and academically, has helped me during all difficult years of my studies. Without their support I would not be here!

To My Parents:

Maryam Azarboo

&

Mohammad Cheshmi

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Sparse matrix computations are an important class of algorithms frequently used in scientific simulations and numerical optimization. Sparse matrix kernels often dominate the overall execution time of many simulations. The performance and efficient memory usage of these simulations and algorithms depend heavily on their use of specialized sparse matrix data structures that only store the nonzero entries. However, such compaction is done using index arrays that result in indirect array accesses such as `A[B[i]]` where `A` and `B` are both arrays. Compilers only apply a limited set of optimizations to sparse code because of the indirection from indexing and looping over the nonzero elements of a sparse data structure. Numerical libraries can provide high-performance code for sparse kernels however they must be manually tuned and optimized for different inputs and architectures, and their performance stagnates with hardware advances.

This dissertation presents a set of code transformations and algorithms, all implemented in a novel code generator called Sympiler, that automates the optimization of sparse matrix codes on parallel processors. Sympiler takes a sparse method, arising from a sparse linear system or sparse numerical optimization, and decouples information related to the computation pattern of the method, i.e., symbolic information, and uses this information to transform the code to vectorizable and parallel code. Sympiler also enables the reuse of symbolic information when the computation pattern remains static for a period of time in the simulations or for when it changes modestly. These optimization techniques combined lead to Sympiler generating a code that is faster than that of hand-written and highly optimized libraries. Because Sympiler is a code generator, similar to compilers, it requires programming and tuning efforts from the user. This chapter provides background and motivation for the research presented in this dissertation and explains the solved challenges and the proposed contributions.

1.1 Sparse Matrix Computations

Sparse matrix kernels are found in a large class of numerical methods as well as optimization algorithms. Numerical methods that solve linear systems of equations are classified as direct and indirect (iterative) methods. Each of these solver classes solves linear system $Ax = b$ to find unknown vector x where A is often a large sparse matrix, and b is the right-hand side vector. Direct sparse linear solvers directly decompose matrix A. The decomposition is done using a factorization algorithm such as LU and Cholesky [71], and then the computed factors are used to find the final solution x in a solve phase which typically involves a sparse triangular solve operation. The indirect class of sparse linear solvers uses an iterative scheme to find the solution and they often require factorizing a sparse matrix, e.g. for their preconditioning, and also computing numerous matrix-vector multiplications to for example update the residual. Optimization methods also frequently operate on sparse matrices. For example, sparse quadratic programming algorithms in each iteration require solving a sparse linear system of equations.

Due to the pivotal role of sparse matrix computations in many numerical and optimization methods, a large class of prior work has attempted to accelerate their execution. A sparse matrix is defined as a matrix in which the majority of its entries are zero. To efficiently store a sparse matrix, it is typically stored in a compressed form such as a compressed row or a compressed column storage format where nonzero elements are stored consecutively row-wise or column-wise with an extra array to store the beginning of the rows or columns. To operate on the sparse matrix, this compact form is accessed. While this compaction will lead to efficient storage, it creates numerous challenges in optimizing the sparse matrix computations. For example, data dependencies between operations and accesses are challenging to track but are necessary to know for running sparse code in parallel. Also, using the memory hierarchy of the processor efficiently is challenging due to compact and irregular indexing in the code.

1.2 Current Solutions to Accelerate Sparse Matrix Computations and Limitations

The most common approach to accelerating sparse matrix computations is to identify a specialized library that provides a manually-tuned implementation of the specific sparse matrix routine. A large number of sparse libraries are available (e.g., SuperLU [44], MUMPS [6], CHOLMOD [26], KLU [42], UMFPACK [35]) for different

numerical kernels, supported architectures, and specific kinds of matrices. Parallel sparse libraries, such as Intel's Math Kernel Library (MKL) [192], Pardiso [192, 156], PaStiX [86], and SuperLU [111], provide manually-optimized parallel implementations of sparse matrix algorithms and are some of the most commonly-used libraries in simulations using sparse matrices. These libraries use numerical-method-specific code at runtime, during a phase called *symbolic factorization*, to determine data dependencies. Based on this dependence information, different libraries implement different forms of parallelism. For example, PaStiX uses static scheduling of a fine-grained task graph based on empirical measurements of expected runtime for each task; in contrast, MKL Pardiso implements a form of dynamic scheduling for its fine-grained task graph. While hand-written libraries can provide high performance, they must be manually ported to new architectures and may stagnate as architectural advances continue. Alternatively, compilers can be used to optimize code while providing architecture portability due to abstracting architecture details from code transformations.

Compiler loop transformation frameworks such as those based on the polyhedral model use algebraic representations of loop nests to transform code and successfully generate highly efficient parallel dense matrix kernels [10, 105, 145, 185, 181, 25]. However, such frameworks are limited when dealing with non-affine loop bounds and/or array subscripts, both of which arise in sparse codes. Previous work has extended compilers to resolve memory access patterns in sparse codes by building runtime *inspectors* to examine the nonzero structure and using *executors* to transform code execution and implement parallelism [187, 148, 204]. Inspectors use runtime information to build directed acyclic graphs (DAGs) that expose data dependence relations. The DAGs are traversed in topological order to create a list of *level sets* that represent iterations that can execute in parallel; this is known as *wavefront parallelism*. Synchronization between level sets ensures the execution respects data dependencies. However, synchronization between levels in wavefront parallelism can lead to high overheads since the number of levels increases with the DAG critical path. For sparse kernels such as Cholesky with non-uniform workloads, wavefront methods can additionally lead to load imbalance. Also, the wavefront techniques only enable thread-level parallelism and do not improve the performance of each thread, such as by using vectorizations.

The challenges with the current inspector-executor frameworks are as follows:

- The current techniques apply conservative transformations or give up transforming sparse codes due to indirect memory accesses. This leads to inefficient use of the memory hierarchy and does not efficiently explore instruction-level paral-

lelism.

- Available wavefront approaches support parallelism by executing iterations of the sparse kernel in parallel. However, because iterations of a sparse kernel have different workloads, the schedule provided by wavefront techniques often leads to load imbalance. Also, they do not optimize the kernel to improve data locality primarily because their scheduling is based on wavefronts.
- Available inspector-executors (as well as libraries) optimize sparse kernels individually. This leads to load imbalance due to dependencies that exist between two sparse kernels. They also miss data reuse opportunities that exist between kernels.
- It is common that in a scientific simulation, the sparsity patterns of matrices change during simulation. These changes are often incremental and small. Inspector-executor frameworks do not exploit this and require the entire set of iterations to be recomputed for any change in the sparsity patterns.

1.3 The Sympiler Solution

Sympiler is a domain-specific code generator that enables the efficient optimization of a given kernel from specific classes of sparse matrix kernels on single-core and parallel architectures. Sympiler also supports the joint optimization of sparse kernels, and for quadratic problems, efficiently updates the sparse matrix kernel factorization for when the sparsity pattern changes during the solver iterations.

Similar to libraries, the general Sympiler solution does symbolic analysis. However, this analysis is done at compile-time to enable the automatic application of numerous code transformations and optimizations that libraries and compilers cannot apply. The symbolic analysis is conducted using an inspector, and information from the inspection is used to transform an internal representation of the kernel code to generate fast code.

1.3.1 Contributions

This dissertation presents a number of inspection strategies and novel code transformations for sparse computations to solve the list of limitations in Section 1.2 in optimizing sparse matrix kernels. The contributions are:

- A symbolic decoupling strategy along with two inspectors as well as *Inspector-guided transformations* that leverage compile-time information to automatically

optimize sparse matrix codes for a single-core processor. The decoupling strategy and inspectors in Sympiler use runtime information to tile iterations and improve instruction-level parallelism and locality; Sympiler also prunes the iteration space to reduce the number of unnecessary iterations.

- A new Load-Balanced Level Coarsening (LBC) algorithm that inspects sparse kernel data dependence graphs for parallelism while maintaining an efficient trade-off between locality, load balance, and parallelism by coarsening level sets from wavefront parallelism. A novel code transformation, called H-level is also proposed to support running the created schedule of LBC. Also, a novel proportional cost model included in LBC that creates well-balanced partitions for sparse kernels with irregular computations such as sparse Cholesky.
- A Multi-Sparse DAG Partitioner (MSP) algorithm that inspects the data dependence graph of two sparse kernels to create a load-balanced parallel schedule of the fused code with good locality. MSP uses a novel vertex assignment strategy that allows vertices of two sparse kernels to be dispersed throughout execution to improve load balance. Two novel packing strategies in MSP, along with code transformations to enable the packing, that improves data locality within and between the iterations of the two sparse kernels based on a reuse-ratio metric.
- A new sparsity-oriented row modification method, SoMod that enables fast factorization for matrix changes via efficient updates of previously computed factors; SoMod used inside a Quadratic Programming (QP) solver called NASOQ that enables solving large scale sparse quadratic programs. As a result, the factorization in the QP solver does not have to be recomputed for every change in the sparsity. We also propose a new Load-balanced Blocked LDL factorization algorithm (LBL) for the fast, accurate factorization of sparse symmetric indefinite systems when changes are not incremental.

1.3.2 Scope

The proposed algorithms and strategies in Sympiler are tested extensively using a large set of matrices from real-world scientific applications and on a number of sparse kernels used in direct and iterative linear system solvers as well QP solvers. The supported kernels are Cholesky and LDL factorization, triangular solvers, matrix-vector multiplication, Gauss-Seidel, GMRes, incomplete Cholesky, and incomplete LU. However, the Sympiler solution can be applied to other sparse codes with extensions that we plan to address in future work.

1.4 Dissertation Overview

Chapter two provides a general overview of the concepts that are used throughout the dissertation. Chapter three introduces Sympiler and how it decouples symbolic information to transform sparse codes. Chapter four presents LBC and H level transformation for transforming sparse codes into parallel code for multicore processors. Chapter five introduces sparse fusion that fuses two loops with sparse dependencies to improve load balance and locality. Chapter six presents LBL with SoMod, a new solver for indefinite linear systems that enables reusing previous solves when modest changes happen. The same chapter also shows how LBL and SoMod are used in NASOQ, a new quadratic programming solver, and compares with other QP solvers. The final chapter concludes the dissertation and discusses future directions.

Chapter 2

Background

This section of the dissertation provides a brief introduction to how sparse matrices are stored in compact storage formats. It also describes the architecture of multi-core processors and their memory subsystem which is the focus of the dissertation. Finally, the datasets that are used to evaluate the proposed techniques in the dissertations are presented.

$$\begin{bmatrix} 1 & 6 & & & \\ & 2 & & 8 & \\ & & 3 & & \\ & 7 & & 4 & \\ 9 & & & & 5 \end{bmatrix}$$

(a) Sparse Matrix A

Ari	0	0	1	1	2	3	3	4	4
Aci	0	1	1	3	2	1	3	0	4
Ax	1	6	2	8	3	7	4	9	5

(b) Triplet Format

Ap	0	2	4	5	7	9			
Ai	0	1	1	3	2	1	3	0	4
Ax	1	6	2	8	3	7	4	9	5

(c) CSR Format

Ap	0	2	5	6	8	9			
Ai	0	4	0	1	3	2	1	3	4
Ax	1	9	6	2	7	3	8	4	5

(d) CSC Format

Figure 2.1: (a) An example sparse matrix A with 5 rows and 5 columns and 9 nonzero entries. (b) The triplet format of A sorted based on row indices. (c) The Compressed Sparse Row (CSR) format of A. (d) The Compressed Sparse Column (CSC) format of A.

2.1 Sparse Matrix

A sparse matrix is a matrix with most entries zeros. The number of zeros in a matrix depends on the domain and application it came from. An example sparse matrix is shown in Figure 2.1a.

2.1.1 Sparse Matrix Storage Formats

Numerous storage methods and data structures can be used to store a sparse matrix. All data structures use a compact representation to only store nonzero entries. In addition to the compact representation, data structures should be simple and flexible so they can be efficiently used in different applications and matrix operations. This subsection discusses some of the most widely used formats used across this dissertation.

Triplet Form

The triplet format uses three arrays to store every entry with its row and column numbers. Figure 2.1b shows the three arrays of the triplet form corresponding to the matrix shown in Figure 2.1a which are sorted based on row indices. For example, the first nonzero in the first row is stored in `Ax[0]` and its corresponding row and column numbers are in `Ari[0]` and `Aci[0]`.

Compressed Sparse Row (CSR) Format

The CSR format stores the sparse matrix using three arrays. The `Ax` array stores the nonzero elements of the matrix row by row and corresponding column indices to nonzero entries are stored in `Ai`. Each entry `Ap[i]` points to the first value of row `i` in `Ax` and its corresponding column index in `Ai`. For example, Figure 2.1c shows the CSR storage of the sparse matrix in Figure 2.1a. The second nonzero of the second row is stored in `Ax[Ap[1]+1]` which is `Ax[3]` and its column index is `Ai[3]`.

Compressed Sparse Column (CSC) Format

The CSC format stores the nonzero values of the matrix using three arrays. The CSC format follows the same scheme as the CSR format with the difference that values are stored column by column and thus row indices are stored. Each entry in `Ap[i]` points to the location of the first nonzero in column `i` in `Ax` and the locations its corresponding row index in `Ai`. For example Figure 2.1d shows the CSC storage format of matrix in Figure 2.1a. The second nonzero of the fourth column is stored in `Ax[Ap[3]+1]` which is `Ax[7]` and its column index is `Ai[7]`.

2.2 Parallel Architectures

The Flynn classification [60] presents three parallel schemes to process data and instructions on parallel processors: Single Instruction, Multiple Data (SIMD) where

processors run the same instruction on different data; Multiple Instruction, Single Data (MISD) where a set of instructions are applied to the same data; Multiple Instruction, Multiple Data (MIMD) where there are different processors and each executes a set of instructions on different piece of data.

Parallel architectures are also classified into two major classes based on their memory architectures: distributed-memory and shared-memory. In distributed memory systems, each processor has a separate memory space. Accesses to another memory space should be done explicitly in the program. However, processors in a shared memory processor use the same address space and data communication between processors is implicit.

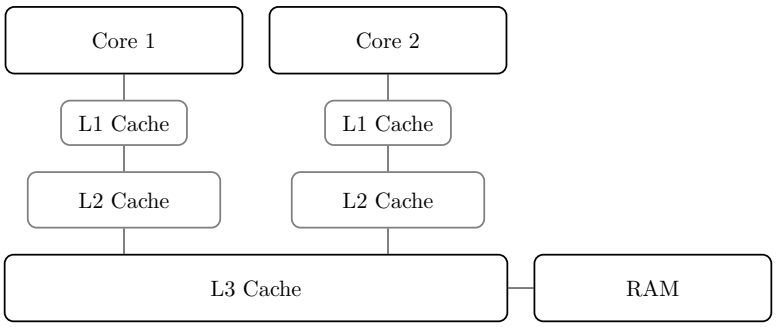


Figure 2.2: The memory hierarchy of a multicore processor with two cores and three levels of cache memories. L1 and L2 caches are dedicated to each processor and L3 is shared between all cores.

2.2.1 Multicore Processors

Multicore processors are used in a wide range of devices including supercomputers, embedded devices, and personal computers. Multicore processors have multiple cores and thus can benefit from MIMD type parallelism. Also, each core typically has vector processors that enable the used of SIMD parallelism as well.

Memory Hierarchy

The central processing unit (CPU) requires access to the main memory to execute instructions. Main memory accesses are several times slower than the CPU clock frequency and recent technological advances have failed to improve the time for such accesses. Instead, hardware platforms have a smaller but cheap-to-access memory space called the cache. By predicting future accesses to the main memory, data can be prefetched into the cache. An example memory hierarchy of a multicore processor with three levels of cache memories is shown in Figure 2.2. As shown the closest cache level to the CPU is the L1 cache which is also the fastest and also smaller than other

Table 2.1: Testbed architectures.

Family	Haswell-E	Haswell-EP	Skylake
Processor	Core™ i7-5820K	Xeon™ E5-2680v3	Xeon™ Platinum 8160
Cores	6 @ 3.30 GHz	12 @ 2.5 GHz	24 @ 2.1 GHz
L3 cache	15MB	30MB	33MB

levels. In Figure 2.2 L1 and L2 caches are dedicated to each core and the last level cache (LLC) is shared amongst all cores.

2.3 Testbed Architectures

All techniques proposed in this dissertation are evaluated on a range of multicore processors listed in Table 2.1. Architectures are selected with different number of cores, i.e. 6, 12, and 24 to evaluate the scalability of the parallel code. At the time of this work, the Haswell-E processor is a standard desktop processor. The other two processors, i.e. Haswell-EP and Skylake are two processors used in Comet and Stampede2 supercomputers. Access to the supercomputers are provided via XSEDE [183].

2.4 Datasets

Throughout the dissertation, all techniques are evaluated on a range of sparse matrices for linear solver applications and also sparse quadratic programs for numerical optimization applications.

2.4.1 Sparse Matrix Dataset

The matrices that are selected to evaluate the proposed techniques in the dissertation are symmetric positive definite (SPD) matrices of different sizes from the SuiteSparse matrix repository [41]. The list of matrices for each technique is provided in the chapter. For example, Chapter 3 uses smaller in size matrices to evaluate the performance of single threaded codes but selected matrices in Chapters 4 and 5 are amongst the largest SPD matrices in the repository to demonstrate how the proposed techniques work for large scale applications.

2.4.2 Sparse Quadratic Programming Dataset

To evaluate the efficiency of the proposed techniques in this dissertation on numerical optimization methods, a set of sparse Quadratic Programming (QP) problems is

collected from real-world applications and with different sizes. The number of variables in the QP problems of the repository is within the range of 50–114309 and their number of constraints are between 20–10k.

Chapter 3

Decoupling Symbolic Information for Code Transformation

For several sparse matrix methods such as LU and Cholesky, it is well known that viewing their computations as a graph (e.g., elimination tree, dependence graph, or quotient graph) and applying a method-dependent graph algorithm yields information about dependencies that can then be used to more efficiently compute the numerical method [37]. Most high-performance sparse matrix computation libraries utilize symbolic information, but couple this symbolic analysis with numeric computation, further making it difficult for compilers to optimize such codes. This chapter presents Sympiler, a domain-specific code generator that produces high-performance sparse matrix code. Sympiler decouples symbolic analysis from numeric computation and transforms the sparse code using symbolic information. The symbolic information is obtained using a symbolic inspector. Inspector-guided transformations, such as variable-sized blocking, are then applied resulting in performance equivalent to hand-tuned libraries. But Sympiler goes further than existing numerical libraries by generating code for a specific matrix nonzero structure. Because the matrix structure often arises from properties of the underlying physical system that the matrix represents, in many cases the same structure reoccurs multiple times, with different values of nonzeros. Thus, Sympiler-generated code can combine inspector-guided and low-level transformations to produce even more efficient code. The transformations applied by Sympiler improve the performance of sparse matrix codes through applying single-core optimizations such as vectorization and increasing data locality. The content of this chapter is published in the conference paper [29].

3.1 Introduction

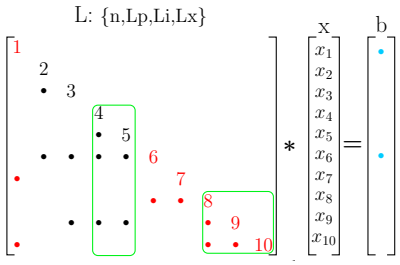
Sparse matrix computations are at the heart of many scientific applications and data analytics codes. The performance and efficient memory usage of these codes depends heavily on their use of specialized sparse matrix data structures that only store the nonzero entries. However, such compaction is done using index arrays that result in indirect array accesses. Due to these indirect array accesses, it is difficult to apply conventional compiler optimizations such as tiling and vectorization even for static index array operations like sparse matrix vector multiply. A static index array does not change during the algorithm; for more complex operations with dynamic index arrays such as matrix factorization and decomposition, the nonzero structure is modified during the computation, making conventional compiler optimization approaches even more difficult to apply.

Libraries and compilers are commonly used to accelerate sparse matrix computations. Hand-written specialized libraries (e.g., SuperLU [44], MUMPS [6], CHOLMOD [26], KLU [42], UMFPACK [35]) provide high performance, but they must be manually ported to new architectures and may stagnate as architectural advances continue. Alternatively, compilers can be used to optimize code while providing architecture portability. Polyhedral compilers [10, 105, 145, 185, 181, 25] use algebraic representations of loop nests to transform code and successfully generate highly-efficient dense matrix kernels. Due to existing non-affine loop bounds and/or array subscripts in sparse codes, run-time *inspectors* [186, 175, 184, 188, 187] extend polyhedral models to examine the nonzero structure and to use *executors* to transform the code to be executed. However, these techniques are limited to transforming sparse kernels with static index arrays.

Sympiler addresses limitations of existing compilers and libraries by performing *symbolic analysis* at compile time to specialize the code for the nonzero pattern whereas the inspector-executor approaches can only reorder data and schedules. Symbolic analysis is a term from the numerical computing community; it uses the nonzero pattern of the sparse matrix to analyze computation patterns. Information from symbolic analysis can be used to make subsequent numeric manipulation faster, and can be reused as long as the matrix nonzero structure remains constant.

3.1.1 Motivating Scenario

The sparse triangular solve takes a lower triangular matrix L and a right-hand side (RHS) vector b and solves the linear equation $Lx = b$ for x. It is a fundamental building block in many numerical algorithms such as factorization [37, 111], direct



(a)

```
x=b; // copy RHS to x
for(j=0;j<n;j++){
  x[j]/=Lx[Lp[j]];
  for(p=Lp[j]+1;p<Lp[j+1];p++){
    x[Li[p]]-=Lx[p]*x[j];}}
```

(b) Forward substitution

```
x=b;
for(j=0;j<n;j++){
  if(x[j] != 0){
    x[j]/=Lx[Lp[j]];
    for(p=Lp[j]+1;p<Lp[j+1];p++)
      x[Li[p]]-=Lx[p]*x[j];}}
```

(c) Library implementation

```
x=b;
for(px=0;px<reachSetSize;px++){
  j=reachSet[px];
  x[j]/=Lx[Lp[j]];
  for(p=Lp[j]+1;p<Lp[j+1];p++){
      x[Li[p]]-=Lx[p]*x[j];}}
```

(d) Decoupled code

```
x=b;
x[0] /= Lx[0]; // Peel col 0
for(p = 1; p < 3; p++)
    x[Li[p]] -= Lx[p] * x[0];
for(px=1;px<3;px++){
  j=reachSet[px];x[j]/=Lx[Lp[j]]
  for(p=Lp[j]+1;p<Lp[j+1];p++)
      x[Li[p]]-=Lx[p]*x[j];}
x[7] /= Lx[20]; // Peel col 7
for(p = 21; p < 23; p++)
    x[Li[p]] -= Lx[p] * x[7];
for(px=4;px<reachSetSize;px++){
  j=reachSet[px];x[j]/=Lx[Lp[j]]
  for(p=Lp[j]+1;p<Lp[j+1];p++)
      x[Li[p]]-=Lx[p]*x[j];}
```

(e) Sympiler-generated

Figure 3.1: Four different codes for solving the linear system in (a). In all four code variants, matrix L is stored in compressed sparse column (CSC) format, with {`n,Lp,Li,Lx`} representing {matrix order, column pointer, row index, nonzeros} respectively. The dependence graph DG_L is the adjacency graph of matrix L; vertices of DG_L correspond to columns of L and its edges show dependencies between columns in triangular solve. Vertices corresponding to nonzero columns are colored in blue and columns that participate in the computation because of the dependence structure are in red. Boxes around columns show supernodes of different sizes. (b) is a forward substitution algorithm. (c) is a library implementation that skips iterations when the corresponding entry in x is zero. (d) is the decoupled code that uses the symbolic information provided by the *reachSet*, which is computed by performing a depth-first search on DG_L. (e) is the Sympiler-generated code which peels iterations corresponding to the columns inside the reach-set with more than 2 nonzeros.

system solvers [34], and rank update methods [38], where the RHS vector is often sparse. A naïve implementation visits every column of the matrix L to propagate the contributions of its corresponding x value to the rest of x (see Figure 3.1b). However, when b is sparse the solution vector is also sparse which can reduce the iteration space of the sparse triangular solve. The reduced iteration space is proportional to the number of nonzero values in x. To benefit from this property, the nonzero pattern of x has to be computed. Based on a theorem from Gilbert and Peierls [66], the *dependence graph* $DG_L = (V, E)$ for matrix L with nodes $V = \{1, ..., n\}$ and edges $E = \{(j, i)|L_{ij} \neq 0\}$ can be used to compute the nonzero pattern of x, where n is the matrix rank and numerical cancellation is neglected. The nonzero indices in x are given by $Reach_L(\beta)$ which is the set of all nodes reachable from any node in $\beta = \{i|b_i \neq 0\}$ and is computed by performing a depth-first search on the directed graph DG_L starting with β. An example dependence graph is illustrated in Figure 3.1a. The blue colored nodes correspond to the set β and the final *reach-set* $Reach_L(\beta)$ contains all the colored nodes.

Figure 3.1 shows four different implementations of sparse triangular solve. All

solvers shown in Figure 3.1 assume the input matrix L is stored in a compressed sparse column (CSC) storage format. While the naïve implementation in Figure 3.1b traverses all columns, the typical library implementation shown in Figure 3.1c skips iterations when the corresponding value in x is zero.

The implementation in Figure 3.1d shows a decoupled code that uses the symbolic information provided by the precomputed reach-set. This decoupling simplifies numerical manipulation and reduces the run-time complexity from $O(|b| + n + f)$ in Figure 3.1c to $O(|b| + f)$ in Figure 3.1d, where f is the number of floating point operations and $|b|$ is the number of nonzeros in b. Sympiler goes further by building the reach-set at compile time and using it to generate code specialized for the specific matrix structure and the RHS. The Sympiler-generated code is shown in Figure 3.1e, where the code only iterates over reached columns and peels iterations where the number of nonzeros in a column is greater than some threshold (in the figure this threshold is 2). The peeling transformation splits some iterations of a loop and performs them outside the body of the loop. These peeled loops can be further transformed with vectorization to speed up execution. This shows the power of fully decoupling the symbolic analysis phase from the code that manipulates numeric values: the compiler aggressively applies conventional optimizations using the reach-set to guide the transformations. On matrices from the SuiteSparse Matrix Collection [41], the Sympiler-generated code shows speedups between 8.4× to 19× with an average of 13.6× compared to the forward solve code (Figure 3.1b) and from 1.2× to 1.7× with an average of 1.3× compared to the library-equivalent code (Figure 3.1c).

3.1.2 Static Sparsity Patterns

Sympiler takes advantage of the fundamental concept that the structure of sparse matrices in scientific codes is dictated by the physical domain and as such does not change in many applications. This structure often arises from the physical topology of the underlying system, the discretization, and the governing equations. These remain unchanged for long periods in simulations across many domains. Some examples include: *(i)* solving nonlinear time-dependent differential equations, where the Jacobian matrix has a static sparsity pattern for each time point while the numerical values change (example domains include fluid dynamics [113] and electromagnetics [118, 48]); *(ii)* design problems where values or parameters are chosen to maximize some measure of performance; (examples include computer animation [13, 146]); *(iii)* domains where the sparse matrix is assembled from a physical topology such as power system modeling; *(iv)* controlling rigid multibody movements in robotics applications

where a sequence of linear systems needs to be solved for a static input [140]; and *(v)* simulations where the sparse stiffness matrix is assembled using a discretized mesh and governing equations and remains static for long periods (e.g. aerospace and electromagnetic simulations).

3.1.3 Contributions

This chapter describes Sympiler, a sparsity-aware code generator for sparse matrix algorithms that leverages symbolic information to generate fast code for a specific matrix structure. Major contributions of this chapter are:

- A novel approach for building *compile-time symbolic inspectors* that obtain information about a sparse matrix for use during compilation.
- *Inspector-guided transformations* that leverage compile-time information to transform sparse matrix code for specific algorithms.
- Implementations of symbolic inspectors and inspector-guided transformations for two algorithms, namely the sparse triangular solve and the sparse Cholesky factorization.
- A demonstration of the performance impact of our code generator, showing that Sympiler-generated code outperforms state-of-the-art libraries for triangular solve and Cholesky factorization by up to 1.7× and 6.3× respectively.

3.2 Sympiler: A Symbolic-Enabled Code Generator

Sympiler generates efficient sparse kernels by tailoring sparse code to specific matrix sparsity structures. By decoupling the symbolic analysis phase, Sympiler uses information from symbolic analysis to guide code generation for the numerical manipulation phase of the kernel. In this section, we describe the overall structure of the Sympiler code generator, as well as the domain-specific transformations enabled by leveraging information from the symbolic inspector.

3.2.1 Sympiler Overview

Sympiler currently supports sparse triangular solve and Cholesky factorization. Given one of these numerical methods and an input matrix stored using the compressed sparse column (CSC) format, Sympiler utilizes a method-specific *symbolic inspector* to obtain information about the matrix. Sample code that uses Sympiler is shown in Figure 3.2a, specifying the input matrix and numerical method. The numerical

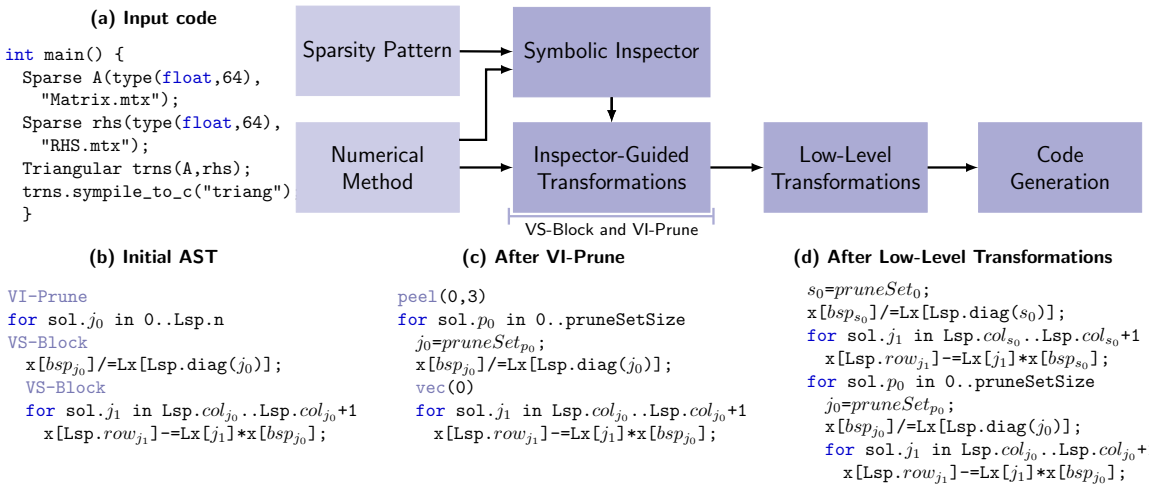


Figure 3.2: Sympiler lowers a functional representation of a sparse kernel to imperative code using the inspection sets. It constructs a set of loop nests and annotates them with domain-specific information that is later used in inspector-guided transformations. The inspector-guided transformations use the lowered code and inspection sets as input and apply transformations. Inspector-guided transformations also provide hints for low-level transformations by annotating the code. For instance, the transformation steps for the code in Figure 3.1 are: (a) Sympiler input code describing input matrices as well as the numerical method; (b) The initial AST with annotations showing where the VI-Prune and VS-Block transformations apply; (c) The transformed code after VI-Prune which has used the pruneSet to add low-level transformation hints such as peeling iterations 0 and 3; (d) The final code where hinted low-level transformations are applied (peeling is only shown for iteration zero).

solver is internally represented using a domain-specific abstract syntax tree (AST) which is annotated with potential transformations. The annotated information is used to apply domain-specific optimizations while lowering the code for the numerical method. Code lowering refers to rewriting code with more details from provided high-level information. In addition, the lowered code is annotated with additional low-level transformations (such as unrolling) when applicable based on domain- and matrix-specific information. Finally, the annotated code is further lowered to apply low-level optimizations and output to C source code.

3.2.2 Symbolic Inspector

Different numerical algorithms can make use of symbolic information in different ways. Prior work has described run-time graph traversal strategies for various numerical methods [142, 112, 33, 37]. The compile-time inspectors in Sympiler are based on these strategies. For each class of numerical algorithms with the same symbolic analysis approach, Sympiler uses a specific symbolic inspector to obtain information about the sparsity structure of the input matrix and stores it in an algorithm-specific way to be used in the transformation stages.

```
1  for(I1){
2   .
3    .
4     for(Ik < m) {
5      .
6       .
7        for(In(Ik,...,In−1)) {
8            a[idx(I1,...,Ik,...,In)];
9        }
10     }
11 }
```

(a) Before

```
1  for(I1) {
2   .
3    .
4     for(Ip < pruneSetSize) {
5      I′k = pruneSet[Ip];
6      .
7       .
8        for(In(I′k,...,In−1)) {
9            a[idx(I1,...,I′k,...,In)];
10       }
11     }
12 }
```

(b) After

Variable Iteration Space Pruning, loop[k].VI-Prune(pruneSet,pruneSetSize)

```
1 for(I) {
2   for(J) {
3     B[idx1(I,J)] op1= a[idx2(I,J)];
4   }
5 }
```

(c) Before

```
1 for(b < blockSetSize) {
2   for(J1 < blockSet[b].x) {
3     for(J2 < blockSet[b].y) {
4       B[idx1(b,J1,J2)] op1 = A[idx2(b, J1, J2)];
5     }
6   }
7 }
```

(d) After

2D Variable-Sized Blocking, loop[I].VS-Block(blockSet,blockSetSize)

Figure 3.3: The inspector-guided transformations. **Top:** The loop over I_k with iteration space m in (a) transforms to a loop over I_p with iteration space *pruneSetSize* in (b). Any use of the original loop index I_k is replaced with its corresponding value from *pruneSet* i.e., I'_k. **Bottom:** The two nested loops in (c) are transformed into loops over variable-sized blocks in (d).

We classify symbolic inspectors based on the numerical method as well as the transformations enabled by the obtained information. For each combination of algorithm and transformation, the symbolic inspector creates an *inspection graph* from the given sparsity pattern and traverses it during inspection using a specific *inspection strategy*. By running the inspector on the inspection graph *inspection sets* are generated. Inspection sets are used to guide transformations in Sympiler.

For our motivating example, triangular solve, the *reach-set* can be used to prune loop iterations that perform work made unnecessary due to the sparsity of the matrix or the right hand side. In this case, the inspection set is the reach-set and the inspection strategy is to perform a depth-first search over the inspection graph, which is the directed dependency graph DG_L of the triangular matrix. For the example linear system shown in Figure 3.1, the symbolic inspector generates the reach-set {6, 1, 7, 8, 9, 10}.

3.2.3 Inspector-guided Transformations

The initial lowered code along with inspection sets obtained by the symbolic inspector go through a series of passes that further transform the code. Sympiler currently

supports two transformations guided by inspection sets, namely *Variable Iteration Space Pruning* and *2D Variable-Sized Blocking*, which can be applied independently or jointly depending on the input sparsity. As shown in Figure 3.2b, the code is internally annotated with information showing where inspector-guided transformations may be applied. The symbolic inspector provides the required information to the transformation phases, which decide whether to transform the code based on the inspection sets. Given the inspection set and annotated code, transformations occur as illustrated in Figure 3.3.

Variable Iteration Space Pruning

Variable Iteration Space Pruning (VI-Prune) prunes the iteration space of a loop using information about the sparse computation. The iteration space for sparse codes can be considerably smaller than that of dense codes, since only iterations with nonzeros are computed. The inspection stage of Sympiler generates an inspection set that enables transforming the unoptimized sparse code to a code with a reduced iteration space.

Given this inspection set, VI-Prune is applied at a particular loop-level to transform the code from Figure 3.3a to Figure 3.3b. In the figure, the transformation is applied to the k^{th} loop nest in line 4. In the transformed code the iteration space is pruned to `pruneSetSize`, which is the inspection set size. In addition to the new loop, all references to I_k are replaced by its corresponding value from the inspection set, `pruneSet[`I_p`]`. Furthermore, the transformation phase uses the inspection set information to annotate certain loops with low-level optimizations. These low-level transformations are applied in subsequent stages of code generation and are guided by tunable thresholds to generate faster code.

In our running triangular solve example, the VI-Prune transformation elides unnecessary iterations due to zeros in the right hand side. In addition, depending on the number of iterations the loops will run (which is known thanks to the symbolic inspector), loops are annotated with directives to unroll and/or vectorize during code generation.

2D Variable-Sized Blocking

2D Variable-Sized Blocking (VS-Block) converts a sparse code to a set of non-uniform dense sub-kernels. In contrast to the conventional approach of blocking/tiling dense codes, where the input and computations are blocked into smaller uniform sub-kernels, the unstructured computations and inputs in sparse kernels make blocking optimizations challenging. The symbolic inspector identifies sub-kernels with similar structure

in the sparse matrix methods and sparse inputs to find "blockable" sets that are not necessarily of the same size or consecutively located. These blocks are similar to the concept of *supernodes* [111] in sparse libraries. VS-Block must deal with a number of challenges:

- The block sizes are variable in a sparse kernel.
- Because of using compressed storage formats, the block elements may not be in consecutive memory locations.
- The type of numerical method used may have to change after applying this transformation. For example, to apply VS-Block to sparse Cholesky code, a dense Cholesky factorization has to be applied to the diagonal segment of the blocks and the off-diagonal segments need to be updated with dense triangular solves.

To address the first challenge, the symbolic inspector provides an inspection set which specifies the size of each block. For the second challenge, the transformed code allocates temporary block storage and copies data as needed prior to operating on the block. Finally, to deal with the last challenge, the synthesized loops/instructions in the lowering phase will contain information about the block location in the matrix and the correct operation is chosen for each loop/instruction when applying the transformation. Similar to VI-Prune, VS-Block also annotates loops with low-level transformations such as tiling for the code generation phase. By leveraging specific information about the matrix when applying the transformation, Sympiler is able to apply VS-Block to sparse numerical methods.

An off-diagonal version of the VS-Block transformation is shown in Figures 3.3c and 3.3d. A new outer loop is created. This outer loop provides block information to the inner loops using the *blockSet*. The inner loop in Figure 3.3c is transformed to two nested loops (lines 2–6) that iterate over the block specified by the outer loop. In line 3 of Figure 3.3c, the vector a is operated on and the result is stored in matrix B. After the VS-Block transformation, this vector operation is converted to a matrix-matrix operation as shown in line 4 of Figure 3.3d. Indices J_1 and J_2 are used to access a particular block in matrix A. Examples of applying VS-Block to triangular solve and Cholesky factorization are provided in Section 3.3.

3.2.4 Enabled Conventional Low-level Transformations

While applying inspector-guided transformations, the original loop nests are transformed into new loops with potentially different iteration spaces, enabling the application of conventional low-level transformations. Based on the applied inspector-guided

transformations as well as the properties of the input matrix and the right-hand side vectors, the code is annotated with transformation directives. An example of these annotations is shown in Figure 3.2c where loop peeling is annotated within the VI-Pruned code. To decide when to add these annotations, the inspector-guided transformations use sparsity-related parameters such as the average block size. Following lists sources that enable Low-level transformations:

1. Symbolic information provides dependency information at compile time allowing Sympiler to apply more transformations such as peeling based on the reach-set in Figure 3.1;

2. Inspector-guided transformations remove some of the indirect memory accesses and annotate the code with potential conventional transformations;

3. Sparsity-specific code generation provides Sympiler with information such as loop boundaries at compile time. A a result, several customized transformations are applied such as vectorization of loops with iteration counts greater than a threshold.

Figure 3.1e demonstrates the process in which iterations in the triangular solve code after VI-Prune are peeled. In this example, the inspection set used for VI-Prune is the reach-set $\{1, 6, 8, 9, 10\}$. Because the reach-set is created in topological order, iteration ordering dependencies are met and thus code correctness is guaranteed after loop peeling. As shown in Figure 3.2c, the transformed code after VI-Prune is annotated with the enabled peeling transformation based on the number of nonzeros in the columns (the *column count*). The two selected iterations with column count greater than two are peeled. The peeled iterations are either replaced with a specialized kernel or another transformation such as vectorization is applied to them.

3.3 Case Studies

Table 3.1: Inspection and transformation elements in Sympiler for triangular solve. DG: dependency graph, SP (RHS): sparsity patterns of the right-hand side vector, DFS: depth-first search, unroll: loop unrolling, peel: loop peeling, dist: loop distribution, tile: loop tiling.

Transformations	**Triangular Solve**			
	Inspection Graph	Inspection Strategy	Inspection Set	Enabled Low-level
VI-Prune	DG + SP(RHS)	DFS	Prune-set (reach-set)	dist, unroll, peel, vectorization,
VS-Block	DG	Node equivalence	Block-set (supernodes)	tile, unroll, peel, vectorization

Table 3.2: Inspection and transformation elements in Sympiler for Cholesky factorization. SP(A): sparsity patterns of the coefficient A, SP (L_j): sparsity patterns of the j^{th}row of L, unroll: loop unrolling, peel: loop peeling, dist: loop distribution, tile: loop tiling.

Transformations	**Cholesky Factorization**			
	Inspection Graph	Inspection Strategy	Inspection Set	Enabled Low-level
VI-Prune	etree + SP(A)	Single-node up-traversal	Prune-set (SP(L_j))	dist, unroll, peel, vectorization
VS-Block	etree + ColCount(L)	Up-traversal	Block-set (supernodes)	tile, unroll, peel, vectorization

Sympiler currently supports two important sparse matrix computations, namely the triangular solve and Cholesky factorization. This section discusses some of the graph theory and algorithms used in Sympiler's symbolic inspector to extract inspection sets for these two matrix methods. The run-time complexity of the symbolic inspector is also presented to evaluate inspection overheads. Finally, we demonstrate the process of applying the VI-Prune and VS-Block transformations using the inspection sets. Sympiler's extension to other matrix methods is also discussed.

Tables 3.1 and 3.2 show a classification of the inspection graphs, inspection strategies, and resulting inspection sets for the two studied numerical algorithms in Sympiler. As shown in both Tables 3.1 and 3.2, the symbolic inspector performs a set of known inspection methods and generates sets which include symbolic information. The last column of the Tables shows the list of low-level transformations enabled by each inspector-guided transformation.

3.3.1 Sparse Triangular Solve

Theory: The symbolic inspector traverses the dependency graph DG_L using depth-first search (DFS) to determine the inspection set for the VI-Prune transformation, which in this case is the reach-set from DG_L and the right-hand side vector. The graph DG_L is also used to detect blocks with similar sparsity patterns, also known as supernodes, in sparse triangular solve. The block-set, which contains columns of L grouped into supernodes, is identified by inspecting DG_L using a node equivalence method. The node equivalence algorithm initially assumes nodes v_i and v_j are equivalent and compares their outgoing edges. If the outgoing edges point to the same destination nodes then the two nodes are equal and are merged.

Inspector-guided Transformations: Using the reach-set, VI-Prune limits the iteration space of the loops in triangular solve to only those that operate on relevant nonzeros. The VS-Block transformation changes the loops in triangular solve to apply blocking as shown in Figure 3.2b. The diagonal block of each column-block, which

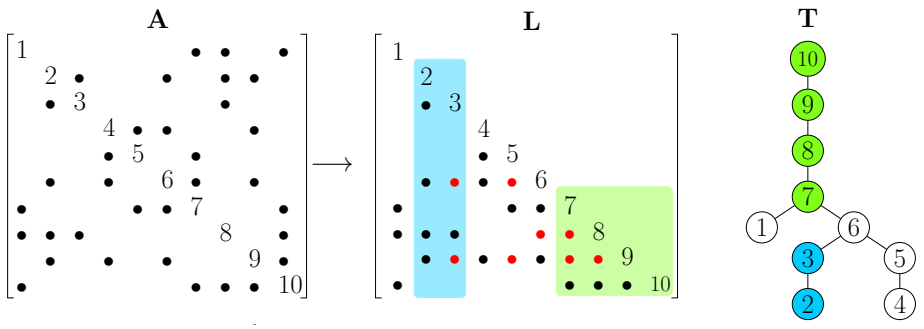


Figure 3.4: An example matrix A and its L factor from Cholesky factorization. The corresponding elimination tree (T) of A is also shown. Nodes in T and columns in L highlighted with the same color belong to the same supernode. The red nonzeros in L are fill-ins.

```
1  for(column j = 0 to n){
2    f = A(:,j)
3    PruneSet = The sparsity pattern of row j
4    for(every row r in PruneSet){ // Update
5        f -= L(j:n,r) * L(j,r);
6    }
7    L(k,k) = sqrt(f(k)); // Diagonal
8    for(off-diagonal elements in f){ // Off-diagonal
9      L(k+1:n,k) = f(k+1:n) / L(k,k);
10   }
11 }
```

Figure 3.5: Pseudo-code of left-looking Cholesky.

is a small triangular solve, is solved first and its solution replaces the off-diagonal segment of the matrix.

Symbolic Inspection: The time complexity of DFS on graph DG_L is proportional to the number of edges traversed and the number of nonzeros in the RHS of the system. The time complexity for the node equivalence algorithm is proportional to the number of nonzeros in L. We provide overheads for these methods for the tested matrices in Section 3.4.3.

3.3.2 Cholesky Factorization

Cholesky factorization is commonly used in direct solvers and is used to precondition iterative solvers. The algorithm factors a Hermitian positive definite matrix A into LL^T, where matrix L is a lower triangular matrix. Figure 3.4 shows an example matrix A and the corresponding L matrix after factorization.

Theory: The elimination tree (etree) [40] is one of the most important graph structures used in the symbolic analysis of sparse factorization algorithms. Figure 3.4 shows the corresponding elimination tree for factorizing matrix A. The etree of A is a spanning tree of $G^+(A)$ satisfying $parent[j] = min\{i > j : L_{ij} \neq 0\}$ where

$G^+(A)$ is the graph of $L + L^T$. The filled graph or $G^+(A)$ results at the end of the elimination process and includes all edges of the original matrix A as well as the fill-in edges. Detailed discussions of the theory behind the elimination tree, the elimination process, and the filled graph can be found in [37, 142].

Figure 3.5 shows the pseudo-code of the left-looking sparse Cholesky, which is performed in two phases of *update* (lines 3–6) and *column factorization* (lines 7–10). The update phase gathers the contributions from the already factorized columns on the left. The column factorization phase calculates the square root of the diagonal element and applies it to the off-diagonal elements.

To find the prune-set that enables the VI-Prune transformation, the row sparsity pattern of L has to be computed. Figure 3.5 shows how this information is used to prune the iteration space of the update phase in the Cholesky algorithm. Since L is stored in column compressed format, the etree and the sparsity pattern of A are used to determine the L row sparsity pattern. A non-optimal method for finding the row sparsity pattern of row i in L is that for each nonzero A_{ij} the etree of A is traversed upwards from node j until node i is reached or a marked node is found. The row-count of i is the visited nodes in this subtree. Sympiler uses a similar but more optimized approach from [37] to find row sparsity patterns.

Supernodes used in VS-Block for Cholesky are found with the L sparsity pattern and the etree. The sparsity pattern of L is different from A because of fill-ins created during factorization. However, the elimination tree T along with the sparsity pattern of A are used to find the sparsity pattern of L prior to factorization. As a result, memory for L is allocated ahead of time to eliminate the need for dynamic memory allocation. To create the supernodes, the fill-in pattern should be first determined. Equation (3.1) is based on a theorem from [61] and computes the sparsity pattern of column j in L, L_j, where $T(s)$ is the parent of node s in T and "\" means exclusion. The theorem states that the nonzero pattern of L_j is the union of the nonzero patterns of the children of j in the etree and the nonzero pattern of column j in A.

$$L_j = A_j \bigcup \{j\} \bigcup \left(\bigcup_{j=T(s)} L_s \backslash \{s\} \right) \tag{3.1}$$

When the sparsity pattern of L is obtained, the following rule is used to merge columns to create basic supernodes: when the number of nonzeros in two adjacent columns j and $j-1$, regardless of the diagonal entry in $j-1$ are equal and $j-1$ is the only child of j in T, the two columns are merged.

Inspector-guided transformations: The VI-Prune transformation is applied to the update phase of Cholesky. With the row sparsity pattern information when

factorizing column i, Sympiler only iterates over dependent columns instead of all columns smaller than i. The VS-Block transformation can be applied to both the update and the column factorization phases. Therefore, the outer loop in the Cholesky algorithm in Figure 3.5 is converted to a new loop that iterates over the block-set. All references to the column j in the inner loops will be changed to the $blockSet[j]$. For the diagonal part of the column factorization, a dense Cholesky needs to be computed instead of the square root in the non-supernodal version. The resulting factor from the diagonal elements applies to the off-diagonal rows through a sequence of dense triangular solves. VS-Block also converts the update phase from vector operations to matrix operations.

Symbolic Inspection: The computational complexity for building the etree in sympiler is nearly $O(|A|)$. The run-time complexity for finding the sparsity pattern of row i is proportional to the number of nonzeros in row i of A. The method is executed for all columns which results in a run-time of nearly $O(|A|)$.The inspection overhead for finding the block-set for VS-Block includes the sparsity detection which is done in nearly $O(|A| + 2n)$ and the supernode detection which has a run-time complexity of $O(n)$ [37].

3.3.3 Other Matrix Methods

The inspection graphs and inspection strategies supported in the current version of Sympiler are used in a large class of commonly-used sparse matrix computations. The applications of the elimination tree go beyond the Cholesky factorization method and extend to some of the most commonly used sparse matrix routines in scientific applications such as LU, QR, orthogonal factorization methods [116], and incomplete and factorized sparse approximate inverse preconditioner computations [97]. Inspection of the dependency graph and proposed inspection strategies that extract reach-sets and supernodes from the dependency graph are the fundamental symbolic analyses required to optimize algorithms such as rank update/downdate methods [38], incomplete LU(0) [131], incomplete Cholesky preconditioners, and up-looking implementations of factorization algorithms. Thus, Sympiler with the current set of symbolic inspectors can be made to support many of these matrix methods. However, Sympiler's decoupling technique is not applicable to methods that change the sparsity pattern during computation such as numerical methods with pivoting. Numerical methods that use sparsity-preserving pivoting techniques such as Bunch-Kaufman [154] or static pivoting [92] can still benefit from Sympiler. We plan to extend to an even larger class of matrix methods and to support more optimization techniques.

Table 3.3: Matrix set: The matrices are sorted based on the number of nonzeros in the original matrix; *nnz* refers to number of nonzeros, n is the rank of the matrix.

Problem ID	Name	n (10^3)	nnz (A) (10^6)
1	cbuckle	13.7	0.677
2	Pres_Poisson	14.8	0.716
3	gyro	17.4	1.02
4	gyro_k	17.4	1.02
5	Dubcova2	65.0	1.03
6	msc23052	23.1	1.14
7	thermomech_dM	204	1.42
8	Dubcova3	147	3.64
9	parabolic_fem	526	3.67
10	ecology2	1000	5.00
11	tmt_sym	727	5.08

3.4 Experimental Results

We evaluate Sympiler by comparing the performance to two state-of-the-art libraries, namely Eigen [79] and CHOLMOD [26], for the Cholesky factorization method and the sparse triangular solve algorithm. Section 3.4.1 discusses the experimental setup and experimental methodology. In Section 3.4.2 we demonstrate that the transformations enabled by Sympiler generate highly-optimized codes for sparse matrix algorithms compared to state-of-the-art libraries. Although symbolic analysis is performed only once at compile time for a fixed sparsity pattern in Sympiler, we analyze the cost of the symbolic inspector in Section 3.4.3 and compare it with symbolic costs in Eigen and CHOLMOD.

3.4.1 Methodology

We selected a set of symmetric positive definite matrices from [41], which are listed in Table 3.3. The matrices originate from different domains and vary in size. All matrices have real numbers and are in double precision. The testbed architecture is Haswell-E as described in Table 2.1 with turbo-boost disabled. OpenBLAS.0.2.19 [198] is used for dense BLAS (Basic Linear Algebra Subprogram) routines when needed. All Sympiler-generated codes are compiled with GCC v.5.4.0 using the `-O3` option. Each experiment is executed 5 times and the median is reported.

We compare the performance of the Sympiler-generated code with CHOLMOD [26] as a specialized library for Cholesky factorization and with Eigen [79] as a general numerical library. CHOLMOD provides one of the fastest implementations of Cholesky factorization on single-core architectures [76]. Eigen supports a wide range of sparse and dense operations including sparse triangular solve and Cholesky. Thus, Cholesky factorization results are compared with both Eigen and CHOLMOD while results for

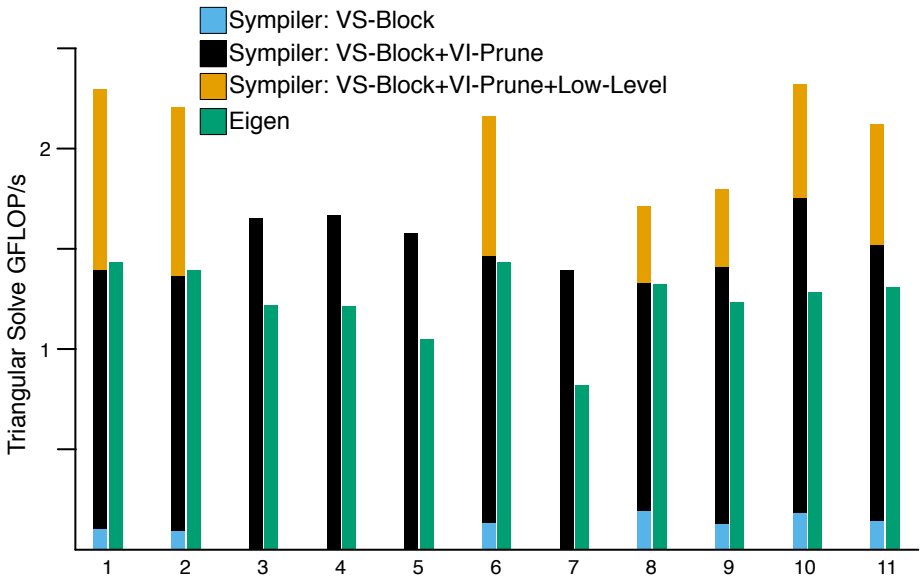


Figure 3.6: Sympiler's performance compared to Eigen for triangular solve. The stacked-bars show the performance of the Sympiler (numeric) code with VS-Block and VI-Prune. The effects of VS-Block, VI-Prune, and low-level transformations on Sympiler's performance are shown separately.

triangular solve are compared to Eigen. Both libraries are installed and executed using the recommended default configuration. For the Cholesky factorization both libraries support the more commonly used left-looking (supernodal) algorithm which is also the algorithm used by Sympiler. Sympiler applies one or both of the inspector-guided transformations as well as some of the enabled low-level transformations. For low-level transformations, Sympiler currently supports unrolling, scalar replacement, and loop distribution. For direct comparison of different implementations, a constant floating point operation (FLOP) count is used across all implementations.

3.4.2 Performance of Generated Code

This section shows how the combination of the introduced transformations and the decoupling strategy enable Sympiler to outperform two state-of-the-art libraries for sparse Cholesky and sparse triangular solve.

Triangular solve: Figure 3.6 shows the performance of Sympiler-generated code compared to the Eigen library for a sparse triangular solve with a sparse RHS. The nonzero fill-in of the RHS in our experiments is selected to be less than 5%. The sparse triangular system solver is often used as a sub-kernel in algorithms such as left-looking LU [37] and Cholesky rank update methods [38] or as a solver after matrix factorizations. Thus, typically the sparsity of the RHS in sparse triangular systems is close to the sparsity of the columns of a sparse matrix. For the tested problems, the number of nonzeros for all columns of L is less than 5%.

The average improvement of Sympiler-generated code, which we refer to as Sympiler (numeric), over the Eigen library is 1.49×. Eigen implements the approach demonstrated in Figure 3.1c, where symbolic analysis is not decoupled from the numerical code. However, the Sympiler-generated code only manipulates numerical values which leads to higher performance. Figure 3.6 also shows the effect of each transformation on the overall performance of the Sympiler-generated code. In the current version of Sympiler the symbolic inspector is designed to generate sets so that VS-Block can be applied before VI-Prune. Our experiments show that this ordering often leads to better performance mainly because Sympiler supports supernodes with a full diagonal block. As support for more transformations are added to Sympiler, we will enable it to automatically decide the best transformation ordering. Whenever applicable, vectorization and peeling transformations are applied after VS-Block and VI-Prune. Peeling leads to higher performance if applied after VS-Block where iterations related to single-column supernodes are peeled. Vectorization is always applied after VS-Block and does not improve performance if only VI-Prune is applied.

Matrices 3, 4, 5, and 7 do not benefit from the VS-Block transformation so their Sympiler run-times in Figure 3.6 are only for VI-Prune. Since small supernodes often do not lead to better performance, Sympiler does not apply the VS-Block transformation if the average size of the participating supernodes is smaller than a threshold. This parameter is currently hand-tuned and is set to 160. VS-Block is not applied to matrices 3, 4, 5, and 7 since the average supernode size is too small and thus does not improve performance. Also, since these matrices have a small column count vectorization does not payoff.

Cholesky: We compare the numerical manipulation code of Eigen and CHOLMOD for Cholesky factorization with the Sympiler-generated code. The results for CHOLMOD and Eigen in Figure 3.7 refer to the numerical code performance in floating point operations per second (FLOP/s). Eigen and CHOLMOD both execute parts of the symbolic analysis only once if the user explicitly indicates that the same sparse matrix is used for subsequent executions. However, even with such an input from the user, none of the libraries fully decouple the symbolic information from the numerical code. This is because they cannot afford to have a separate implementation for each sparsity pattern and also do not implement sparsity-specific optimizations. For fairness, when using Eigen and CHOLMOD we explicitly tell the library that the sparsity is fixed and thus report only the time related to the library's numerical code (which still contains some symbolic analysis).

As shown in Figure 3.7, for Cholesky factorization Sympiler performs up to 1.3×, 2.3×, and 6.3× better than CHOLMOD with node amalgamation [50], CHOLMOD

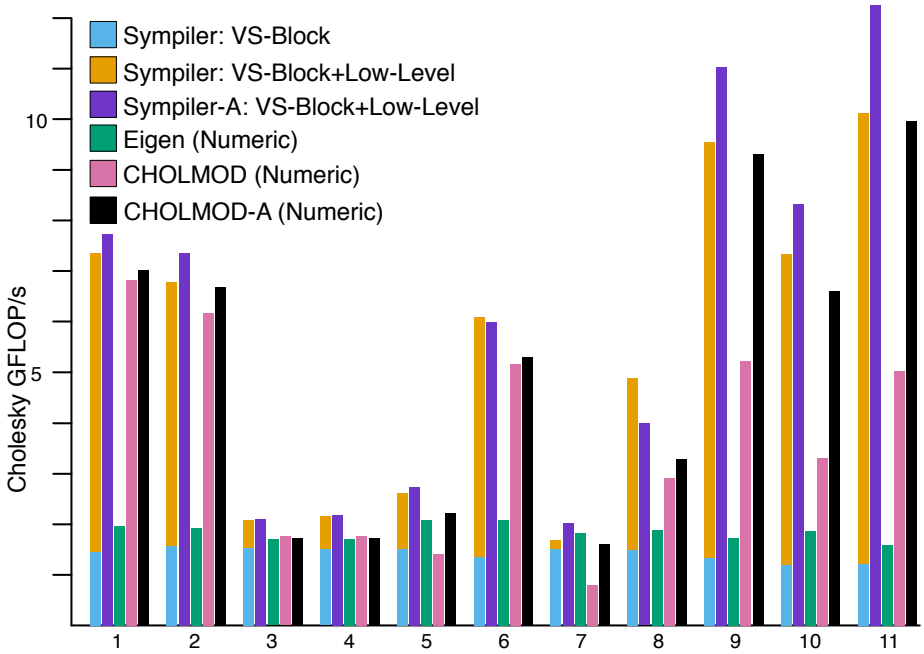


Figure 3.7: The performance of Sympiler (numeric) for Cholesky compared to CHOLMOD (numeric) and Eigen (numeric). The stacked-bar shows the performance of the Sympiler-generated code. The effect of VS-Block and low-level transformations are shown separately. The VI-Prune transformation is already applied to the baseline code so it is not shown here. Sympiler-A and CHOLMOD-A refer to versions with node amalgamation.

without node amalgamation, and Eigen respectively. Eigen uses the left-looking non-supernodal algorithm and thus its performance does not scale well with large matrices. CHOLMOD benefits from supernodes and performs well for large matrices with large supernodes. However, CHOLMOD does not perform well for some small matrices and large matrices with small supernodes. Node amalgamation merges small supernodes to increase their size for better performance and is implemented in both CHOLMOD and Sympiler. Sympiler provides the highest performance for almost all tested matrix types which demonstrates the effectiveness of sparsity-specific code generation.

The application of kernel-specific and aggressive optimizations when generating code for dense sub-kernels enables Sympiler to generate fast code for any sparsity pattern. Since BLAS routines are not well-optimized for small dense kernels they often do not perform well for the small blocks produced by applying VS-Block to sparse codes [163]. Therefore, libraries such as CHOLMOD do not perform well for matrices with small supernodes. Sympiler has the luxury to generate code for its dense sub-kernels; instead of being handicapped by the performance of BLAS routines, it generates specialized and highly-efficient codes for small dense sub-kernels. If the average column-count for a matrix is above a tuned threshold, Sympiler will call BLAS routines [198] instead. Since the column-count directly specifies the number

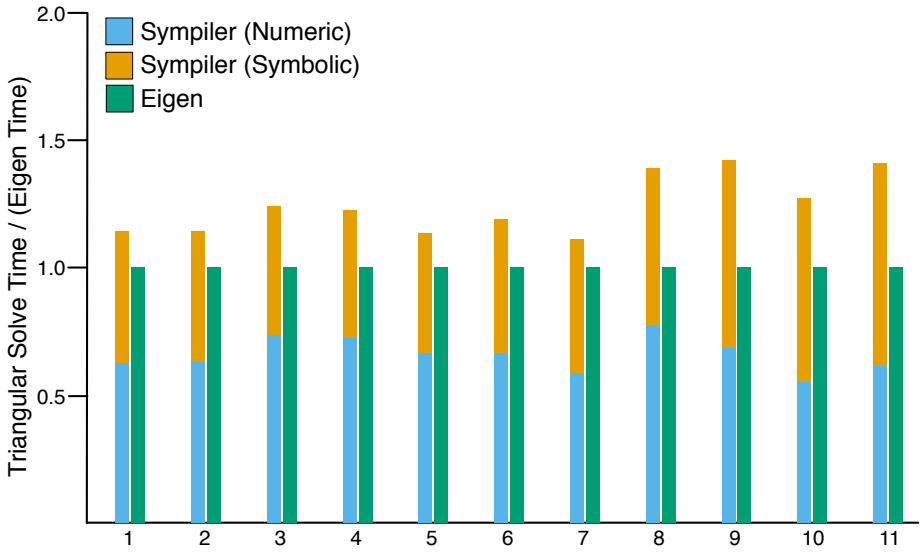


Figure 3.8: Sparse triangular solve symbolic+numeric time for Sympiler and Eigen's normalized over the Eigen time.

of dense triangular solves, which is the most important dense sub-kernel in Cholesky, the average column-count is used to decide when to switch to BLAS routines [198]. For example, the average column-count of matrices 3, 4, 6, and 8 is less than the column-count threshold.

Decoupling the prune-set calculation from the numerical manipulation phase also improves the performance of the Sympiler-generated code. As discussed in subsection 3.3.2, the sparse Cholesky implementation needs the row sparsity pattern of L. The elimination tree of A and the upper triangular part of A are both used in CHOLMOD and Eigen to find the row sparsity pattern. Since A is symmetric with only the lower part stored, both libraries compute the transpose of A in the numerical code to access the upper triangular elements. Through fully decoupling symbolic analysis from the numerical code, Sympiler has the L row sparsity information in the prune-set ahead of time. Therefore, both the reach function and the matrix transpose operations are removed from the numeric code.

3.4.3 Symbolic Analysis Time

All symbolic analysis is performed at compile time in Sympiler and its generated code only manipulates numerical values. Since symbolic analysis is performed once for a specific sparsity pattern, its overheads amortize with repeat executions of the numerical code. However, as demonstrated in Figures 3.8 and 3.9 even if the numerical code is executed only once, which is not common in scientific applications, the accumulated symbolic+numeric time of Sympiler is close to Eigen for the triangular solve and

faster than both Eigen and CHOLMOD for Cholesky. Since Sympiler also generates generic library code by disabling low-level transformations, the code generation cost is discussed separately and not included in the figures.

Triangular solve: Figure 3.8 shows the time Sympiler spends to do symbolic analysis at compile time, Sympiler (symbolic), for sparse triangular solve, normalized over Eigen's run-time. No symbolic time is available for Eigen since as discussed, Eigen uses the code in Figure 3.1c for its triangular solve implementation. Sympiler's numeric plus symbolic time is on average 1.27× slower than the Eigen code. In addition, depending on the matrix, code generation and compilation in Sympiler costs between 6–197× more than the numeric solve. It is important to note that since the sparsity structure of the matrix in triangular solve does not change in many applications, the overhead of the symbolic inspector and compilation is only paid once. For example, in preconditioned iterative solvers a triangular system must be solved per iteration and often the iterative solver must execute thousands of iterations [18, 136, 106] until convergence since the systems in scientific applications are not necessarily well-conditioned.

Cholesky: Sparse libraries perform symbolic analysis ahead of time which can be re-used for matching sparsity patterns and improves the performance of their numerical executions. We compare the analysis time of the libraries with Sympiler's symbolic inspection time. Figure 3.9 provides the symbolic analysis and numeric manipulation times for both libraries normalized over Eigen time. The time spent by Sympiler to perform symbolic analysis is referred to as Sympiler (symbolic). CHOLMOD (symbolic) and Eigen (symbolic) refer to the partially decoupled symbolic code that is only executed once if the user indicates that sparsity remains static. In nearly all cases Sympiler's accumulated time is better than the other two libraries. Code generation and compilation, which are not shown in the chart, cost at most 0.3× the cost of numeric factorization. Also, similar to the triangular solve example, a matrix with a fixed sparsity pattern must be factorized many times in scientific applications. For example, in Newton-Raphson (NR) solvers for nonlinear systems of equations, a Jacobian matrix is factorized in each iteration and the NR solvers require tens or hundreds of iterations to converge [138, 130].

3.5 Related Work

Compilers for general languages are hampered by optimization methods that either cannot optimize sparse codes or only apply conservative transformations that do not lead to high performance. This is due to the indirection required to index and loop

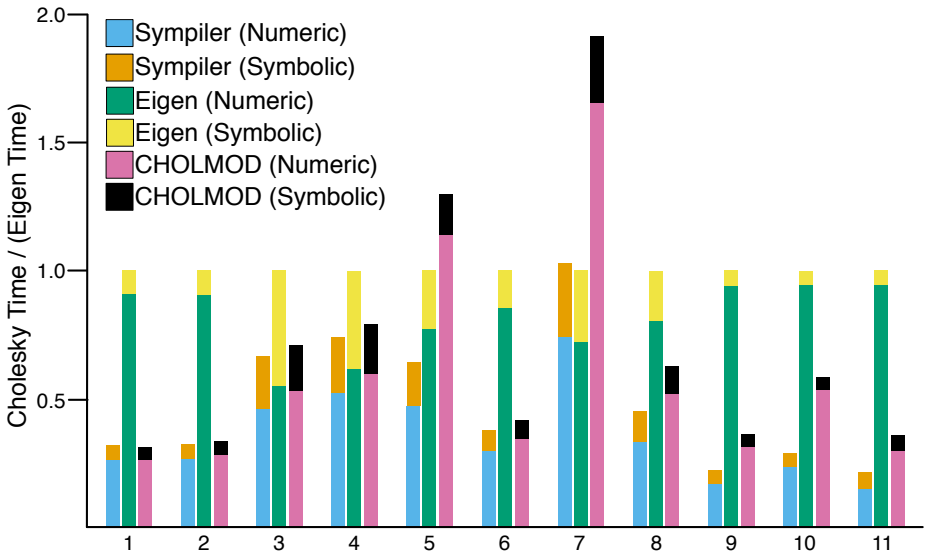


Figure 3.9: Symbolic+numeric time for Sympiler, CHOLMOD, and Eigen for the Cholesky algorithm. All times are normalized over the Eigen's accumulated symbolic+numeric time.

over nonzero elements of sparse data structures. Polyhedral methods use algebraic representations and rules to represent and transform loop nests into optimized code. These techniques are limited when dealing with non-affine loop nests or subscripts [10, 25, 105, 145, 185, 181] common in sparse computations.

To make it possible for compilers to apply more aggressive loop and data transformations to sparse codes, recent work [186, 175, 184, 188, 187] has developed compile-time techniques for automatically creating *inspectors* and *executors* for use at run-time. These techniques use an inspector to analyze index arrays in sparse codes at run-time and an executor that uses this run-time information to execute code with specific optimizations. These inspector-executor techniques are limited in that they only apply to sparse codes with static index arrays; such codes require the matrix structure to not change during the computation. The aforementioned approach performs well for methods such as sparse incomplete LU (0) and Gauss-Seidel methods where additional nonzeros/fill-ins are not introduced during computation. However, in a large class of sparse matrix methods, such as direct solvers including Cholesky, LU, and QR decompositions, index arrays dynamically change during computation since the algorithm itself introduces fill-ins. In addition, the indirections and dependencies in sparse direct solvers are tightly coupled with the algorithm, making it difficult to apply inspector-executor techniques. Partial evaluation techniques [99] specialize code by using partial inputs and the program to generate code that works well for all the remaining inputs. These methods have not been successfully extended to support sparse matrix methods [82].

Domain-specific compilers integrate domain knowledge into the compilation process, improving the compiler's ability to transform and optimize specific kinds of computations. Such an approach has been used successfully for stencil computations [147, 179, 93], signal processing [144], dense linear algebra [80, 169, 115], matrix assembly and mesh analysis [5, 119], simulation [107, 20], and sparse operations [36, 150, 27]. Though the simulations and sparse compilers use some knowledge of matrix structure to optimize operations, they do not build specialized matrix solvers.

Specialized Libraries are the typical approach for sparse direct solvers. These libraries differ in (1) which numerical methods are implemented, (2) the implementation strategy or variant of the solver, (3) the type of the platform supported, and (4) whether the algorithm is specialized for specific applications.

Each numerical method is suitable for different classes of matrices; for example, Cholesky factorization requires the matrix be symmetric (or Hermitian) positive definite. Libraries such as SuperLU [44], KLU [42], UMFPACK [33], and Eigen [79] provide optimized implementations for LU decomposition methods. The Cholesky factorization is available through libraries such as Eigen [79], CSparse [37], CHOLMOD [26], MUMPS [6, 7, 8], and PARDISO [155, 157]. QR factorization is implemented in SPARSPAK[62, 61], SPLOOES [12], Eigen [79], and CSparse [37]. The optimizations and algorithm variants used to implement sparse matrix methods differ between libraries. For example, LU decomposition can be implemented using multifrontal methods [33, 81, 35], left-looking [44, 42, 51, 62], right-looking [112, 161, 49], and up-looking [34, 162] methods. Libraries are developed to support different platforms such as sequential implementations [37, 26, 42], shared memory [35, 45, 155], and distributed memory [45, 7]. Finally, some libraries are designed to perform well on matrices arising from a specific domain. For example, KLU [42] works best for circuit simulation problems. In contrast, SuperLU-MT applies optimizations with the assumption that the input matrix structure leads to large supernodes; such a strategy is a poor fit for circuit simulation problems.

Chapter 4

Transformation and Inspection for Parallelism

Sympiler can generate a specialized code that can effectively use vectorization by decoupling symbolic information as discussed in Chapter 3. However, the Sympiler's generated code runs sequentially and does not use thread-level parallelism. Sympiler's inspectors use runtime information to build directed acyclic graphs (DAGs) that expose data dependence relations. The DAGs are traversed in topological order to create a list of *level sets* that represent iterations that can execute in parallel; this is known as *wavefront parallelism*. Synchronization between level sets ensures the execution respects data dependencies. However, synchronization between levels in wavefront parallelism can lead to high overheads since the number of levels increases with the DAG critical path. For sparse kernels such as Cholesky with non-uniform workloads, wavefront methods can additionally lead to load imbalance. This chapter presents an inspection strategy for parallelism on multi-core architectures for sparse matrix kernels. The proposed inspector applies a novel Load-Balanced Level Coarsening (LBC) algorithm on the data dependence graph to create well-balanced coarsened level sets, which is called the hierarchical level set (H-Level set), mitigating load imbalance and excessive synchronization present in wavefront parallelism. Please note that the content of this chapter is published in the conference paper [28].

4.1 Introduction

The performance of scientific simulations relies heavily on the parallel implementations of sparse matrix computations used to solve systems of linear equations. Data dependence information required for parallelizing sparse codes is dependent on the matrix structure, so parallel codes may use more synchronization than necessary;

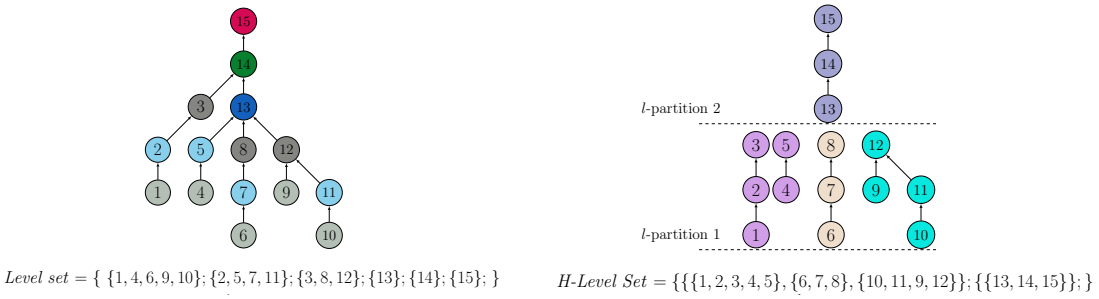
in addition, to achieve high parallel efficiency, the work must be evenly distributed among cores, but this distribution also depends on the matrix structure.

Parallel sparse libraries, such as Intel's Math Kernel Library (MKL) [192], Pardiso [192, 156], PaStiX [86], and SuperLU [111], provide manually-optimized parallel implementations of sparse matrix algorithms and are some of the most commonly-used libraries in simulations using sparse matrices. These libraries differ in the kind of numerical methods they support and use numerical-method-specific code at runtime, during a phase called *symbolic factorization*, to determine data dependencies. Based on this dependence information, different libraries implement different forms of parallelism. For example, PaStiX uses static scheduling of a fine-grained task graph based on empirical measurements of expected runtime for each task; in contrast, MKL Pardiso implements a form of dynamic scheduling for its fine-grained task graph.

Previous work has extended compilers to resolve memory access patterns in sparse codes by building runtime *inspectors* to examine the nonzero structure and using *executors* to transform code execution and implement parallelism [187, 148, 204]. Inspectors use runtime information to build directed acyclic graphs (DAGs) that expose data dependence relations. The DAGs are traversed in topological order to create a list of *level sets* that represent iterations that can execute in parallel; this is known as *wavefront parallelism*. Synchronization between level sets ensures the execution respects data dependencies. However, synchronization between levels in wavefront parallelism can lead to high overheads since the number of levels increases with the DAG critical path. For sparse kernels such as Cholesky with non-uniform workloads, wavefront methods can additionally lead to load imbalance. Frameworks such as Sympiler [29] have demonstrated the value of creating specializations of sparse matrix methods for exploiting specific matrix structure. However, this approach has only been demonstrated for single-threaded implementations.

This chapter presents an inspection strategy for parallelism on multi-core architectures for sparse matrix kernels. The proposed inspector applies a novel Load-Balanced Level Coarsening (LBC) algorithm on the data dependence graph to create well-balanced coarsened level sets, which we call the hierarchical level set (H-Level set), mitigating load imbalance and excessive synchronization present in wavefront parallelism. This inspector is implemented in a framework called ParSy, which uses information from the matrix sparsity and the numerical method to obtain data dependencies. The inspector in ParSy can be used for sparse linear algebra libraries, inspector-executor compiler methods, or from within sparsity-specific code generators such as Sympiler.

We focus on complex sparse matrix algorithms where loop-carried data depen-



(a) Assembly Tree $G(V, E)$

(b) H-Level set of G

Figure 4.1: An example DAG, that is an assembly tree where nodes represent column blocks and edges show the dependencies between columns during factorization, is shown in Figure 4.1a. Wavefront methods create a level set, represented by node coloring; nodes with the same color can be executed in parallel. Figure 4.1b shows the H-Level set created by LBC from G in Figure 4.1a.

dencies make efficient parallelization challenging, such as sparse triangular solve, as well as matrix methods that introduce fill-ins (nonzeros) during computation, such as Cholesky. The main contributions of this chapter include:

- A new Load-Balanced Level Coarsening (LBC) strategy that inspects sparse kernel data dependence graphs for parallelism while maintaining an efficient trade-off between locality, load balance, and parallelism by coarsening level sets from wavefront parallelism.
- A novel proportional cost model included in LBC that creates well-balanced partitions for sparse kernels with irregular computations such as sparse Cholesky.
- Implementations of the new inspection strategies for parallelism and code transformations for sparse triangular solve and Cholesky factorization, in a framework called ParSy. For evaluation, the proposed implementations are built within the open-source Sympiler infrastructure, but with all Sympiler optimizations disabled. The performance of ParSy is evaluated against MKL Pardiso and PaStiX, demonstrating that the partitioning strategy in ParSy outperforms the state-of-the-art by 1.4× on average and up to 3.1×.

4.2 ParSy Overview

ParSy consists of the H-Level inspector and code transformations to enable efficient parallel execution for sparse matrix methods. Example input code to ParSy is shown in Listing 4.1, where the user provides the numerical method, matrix sparsity pattern,

and additional information about the desired level of parallelism. ParSy builds a DAG representing data dependencies in the sparse kernel for the given sparsity pattern. Then, the H-Level inspector uses a Load-Balanced Level Coarsening algorithm to create a schedule from the DAG of the kernel. To parallelize the original code and take advantage of the schedule, the numerical method code must be transformed. This section describes the H-Level inspector and discusses code transformations to support the parallel schedule, using sparse Cholesky as an example.

```
1 int main() {
2  Sparse A(type(float,64),"Matrix.mtx");
3  Cholesky chol(A);
4  chol.generate_c("chol",k); }
```

Listing 4.1: ParSy input code

Algorithm 1: ParSy's H-Level inspector.

Input : DAG G, k, thresh, win, agg
Output: H-LevelSet
1 [vertexCost,edgeCost] = computeCost(G)
2 [H-LevelSet]=LBC(G,vertexCost,edgeCost, k, thresh, win, agg)
3 return H-LevelSet

4.2.1 H-Level Inspector

The goal of ParSy's inspector is to statically partition the DAG of a specific numerical method applied to a specific sparse matrix while creating an efficient load balance with low synchronization cost and high locality. Wavefront parallelism approaches [110, 132], typically used in code transformation frameworks to generate parallel sparse codes, can create load imbalance and excessive synchronizations since sparse kernels like Cholesky have imbalanced workloads for column-based and column-block-based implementations. ParSy's H-Level inspector resolves this issue by creating partitions with coarser tasks while ensuring good balance between execution threads.

Algorithm 1 shows the basic outline of ParSy's inspector. Line 2 shows the LBC phase (see Section 4.3), where the DAG along with the number of processor cores (k in Algorithm 1), the computational efficiency of a single core (*thresh*), and tuning parameters *win* and *agg* related to balancing and coarsening of the levels, are the inputs. The LBC algorithm uses a kernel-specific cost model for vertices and edges, which is used for load balance. With this information the DAG is partitioned into *level-partitions* (l-partitions) that partition the DAG into coarsened levels, and into k or fewer *width-partitions* (w-partitions) each executed on a single core within each l-partition.

Example. Cholesky factorization is commonly used in direct linear solvers and to precondition iterative solvers. The algorithm factors a Hermitian positive definite matrix A into LL^T, where matrix L is a sparse lower triangular matrix. We use the left-looking Cholesky variant. To compute the factor for a column j in L the algorithm visits all columns i that contain a nonzero in row j of L with $i < j$ and then applies the contributions of columns i to column j [37]. Dependencies between each column-computing iteration are represented by a DAG called the elimination tree (etree) [116, 142]. In an etree each node represents a column and each directed edge denotes that the destination depends on the source. To improve the performance of sparse Cholesky by using dense BLAS operations, columns with similar nonzero patterns are merged to form a block or supernode of columns. Dependencies between column blocks are represented using a modified version of the etree called the assembly tree, where nodes represent column blocks. For Cholesky factorization, using the etree does not create coarse enough nodes to parallelize and thus in most available software [44, 86, 158, 6] the assembly tree is used as the baseline dependency DAG for Cholesky. Figure 4.1a is an example assembly tree that we will use to demonstrate how ParSy creates an H-Level set.

Wavefront parallelism techniques [187, 148] first create a topologically-ordered level set, shown in Figure 4.1a and then execute nodes within each level in parallel. However, this often leads to higher-than-necessary overhead, because it requires synchronization between each level. Furthermore, the work per node varies depending on the non-zero structure, often resulting in poor load balance. Our Load-Balanced Level Coarsening (LBC) algorithm, described in Section 4.3, partitions the assembly tree with the objective of facilitating efficient parallel execution while producing a good balance between load and locality. Our partitioning works in two stages; the first partitions the DAG by level to create topologically-ordered l-partitions, shown in Figure 4.1b. In the second phase, the disjoint sub-DAGs inside each level are divided into k or fewer equally-balanced w-partitions, where k is the number of cores. The H-Level set improves locality compared to the wavefront approach and reduces inter-level synchronizations from six to two for this example. Furthermore, the LBC algorithm balances the workload of each partition by packing multiple independent sub-DAGs into each w-partition. This packing approach is important in sparse Cholesky where the workload for each column block differs from other blocks. Finally, each w-partition does not communicate with any other w-partition in the same level, since each w-partition consists of disjoint sub-DAGs.

```
1 H-Level:
2 for(I1){
3    .
4    .
5     .
6
7        for(In(I1)){
8 Atomic:
9            c /= a[idx(I1,...,In)]; }}
```

(a) Before

```
1 for(every l-partition i){
2 #pragma omp parallel for private(pVars)
3     for(every w-partition j){
4       for(every v ∈ HLevelSet[i][j]){
5         I1 = v;
6         ...
7           for(In(I1)){
8 #pragma omp atomic
9               c /= a[idx(I1 ,... ,In)];}}}}}
```

(b) After

Level, loop[1].HLevel(HLevelSet,pVars)

Figure 4.2: The H-Level transformation. The loop over I_1 in (a) transforms into two nested loops that iterate over the `H-Level set` in (b). Any use of the original loop index I_1 is replaced with its corresponding value from `HLevelSet`.

4.2.2 Parallel Code Transformation

To utilize the H-Level set to efficiently execute the schedule, the original code must be transformed for parallelism. Figure 4.2 shows the general form of the H-level transformation. The loop in line 2 of the code in Figure 4.2a is changed to lines 1–4 in the code in Figure 4.2b. After transformation, all operations and indices that use I_1, which is the index of the transformed loop, will be replaced with a proper value from `HLevelSet`. The parallel pragma in line 2 ensures that all w-partitions within an l-partition run in parallel. Note that some algorithms may require atomic pragmas; such cases are detectable using existing analysis techniques [55].

```
1 H-Level:
2 for (int i=0; i<blockNo; ++i){
3   b1 = block2col[i]; b2 = block2col[i+1];
4   f = A(:,b1:b2);
5   // Update phase
6   for(block r=0 to i-1 L(i,r)!=0){
7   f-=GEMM(L(b1:n,r),transpose(L(i,r)));}
8   // Diagonal operation
9   L(b1:b2,b1:b2)=POTRF(f(b1:b2));
10  // Off-diagonal operations
11  for(off-diagonal elements in f){
12  L(b2+1:n,b1:b1) =
13  TRSM(f(b1+1:n,b1:b2),L(b1:b2,b1:b2)); }  }
```

(a) Serial blocked left-looking Cholesky

```
1 for(every l-partition i){
2 #pragma omp parallel for private(f){
3  for(every w-partition j){
4   for(every v ∈ HLevelSet[i][j]){
5     int i = v;
6     b1 = block2col[i];b2 = block2col[i+1];
7     f = A(:,b1:b2);
8     for(block r=0 to i-1 L(i,r)!=0){
9     f-=GEMM(L(b1:n,r),transpose(L(i,r)));}
10    L(b1:b2,b1:b2)=POTRF(f(b1:b2));
11    for(off-diagonal elements in f){
12    L(b2+1:n,b1:b1) =
13    TRSM(f(b1+1:n,b1:b2),L(b1:b2,b1:b2)); } }}}}
14 //Specialized code for the last l-partition.
15 Cholesky_Specialized(HLevelSet[n − 1][0]);
```

(b) ParSy's generated code

Figure 4.3: The application of the H-Level transformation on blocked left-looking Cholesky factorization. Figure 4.3b shows the transformed version of the code in Figure 4.3a with the H-Level transformation. The gray lines remain unchanged.

Figure 4.3 shows how the H-Level transformation modifies Cholesky factorization. As shown, the outermost loop in line 2 of Figure 4.3a is transformed to lines 1–

5 in Figure 4.3b. Since in the left-looking Cholesky algorithm nodes do not write to other nodes, the loop body does not change because no critical region is required. The OpenMP pragma enables parallelism over sub-DAGs, executing dependent nodes within the same thread, which increases locality. For the example DAG in Figure 4.1a, the outer loop in the code of Figure 4.3b executes only one iteration, resulting in a single synchronization, compared to the six synchronizations required by wavefront parallelism.

The available parallelism in a sparse algorithm is not uniform and typically different approaches for parallelism must be used to efficiently exploit the underlying parallel architecture. For example, l-partition 1 in the partitioned DAG in Figure 4.1b benefits from tree parallelism; however, the nodes in l-partition 2, which contains the sink node (the node with no outgoing edges), have no tree parallelism but such nodes can be repartitioned to increase data parallelism within their corresponding dense computations [86]. The last iteration, which corresponds to the last partition of the H-Level set, is peeled and optimized differently. For such nodes, ParSy enables using parallel BLAS operations for the node; however, ParSy can be extended to support more advanced specialization techniques such as repartitioning.

4.2.3 Implementation

We have implemented ParSy in the open-source Sympiler [29] framework. Even though ParSy can be implemented at runtime similar to library-based approaches, we build on top of Sympiler to ease implementation and for potential future benefits of integrating ParSy with sparsity-specific code generation from Sympiler. Because of using Sympiler, the inspectors in ParSy are executed at compile time and their information is used to automatically transform the code.

To implement ParSy, the inputs to Sympiler are extended to provide information that the H-Level inspector requires. The H-Level inspector and H-Level transformation are implemented as additional stages in the inspection and transformation phases of Sympiler respectively. The inspector creates the data dependence graph based on the input numerical method and the sparsity pattern. ParSy uses the created data dependence graph and creates a coarsened level set that will later be used as an input to the generated code. Sympiler's low-level transformations are disabled in ParSy, so we do not specialize code for a specific sparsity pattern. This chapter considers solely the impact of the H-Level inspector.

4.3 Load-Balanced Level Coarsening (LBC)

ParSy utilizes the Load-Balanced Level Coarsening (LBC) algorithm to partition the DAG that describes the dependencies of the computation. LBC statically creates a set of partitions that minimize load imbalance and communication while attempting to maximize available parallelism and locality. In this section, we describe the partitioning produced by LBC, its associated constraints, and the algorithm that produces this partitioning. Finally, we show the proportional cost model used by LBC to estimate load costs for each partition.

4.3.1 Problem Definition

The goal of Load-Balanced Level Coarsening is to find a set of l-partitions, and within each l-partition, to find a set of disjoint w-partitions with as balanced cost as possible. For improved performance, these partitions adhere to additional constraints to reduce synchronization between threads and maintain load balance. Additionally, there are objective functions for minimizing communication between threads and the number of synchronizations between levels. To describe the partitioning and constraints, we use the following notation.

Definitions. $G(V, E)$ denotes the input DAG with vertex set V and edge set E, along with a nonnegative integer weight $d(v)$ for each vertex $v \in V$ and nonnegative integer weight $c(e)$ for each edge $e \in E$. The *level* of a node $level(v)$ is the length of the longest path between the node v and a *source node*, which is a node with no incoming edge. The level of the sink node is the length of the *critical path* P; in the case of multiple sink nodes, P is the maximal level among all sink nodes.

Definition 1: Given DAG G and an integer number of partitions $n > 1$, the LBC algorithm produces n l-partitions of V with sets of nodes $(V_{l_1}, ..., V_{l_n})$ such that $V_{l_1} \cup ... \cup V_{l_n} = V$ and $\forall i \neq j, V_{l_i} \cap V_{l_j} = \emptyset$. Each l-partition $l_i = [lb_i..ub_i]$ is represented by a lower bound and upper bound on the level, and contains all nodes with levels between the two bounds. In addition, $\cup_{i=1}^{n} l_i = [1..P]$. The induced DAG for l-partition l_i is represented with $G_{lb_i:ub_i}$.

Definition 2: Given the number of threads $k > 1$, for each set of nodes V_{l_i}, the LBC algorithm produces $m_i \leq k$ w-partitions $(V_{l_i,w_1}, ..., V_{l_i,w_m})$ such that $V_{l_i,w_1} \cup ... \cup V_{l_i,w_{m_i}} = V_{l_i}$ and $\forall i, j, p, q$, where $i \neq j$ or $p \neq q$, $V_{l_i,w_p} \cap V_{l_j,w_q} = \emptyset$.

Definition 3: Within a partition, the number of *connected components* is the number of disjoint sub-DAGs in the partition, which is shown by $comp(V_{l_i,w_p})$ for a partition V_{l_i,w_p}.

In summary, the partitioning produced by LBC creates l-partitions, and within

each l-partition i, it creates up to k disjoint w-partitions. Each node in the DAG belongs to one l-partition and one w-partition. Note that some l-partitions, those with only one connected component, will only contain one w-partition (see V_{l_2} in Figure 4.1). Some of the values for that example are as follows: $n = 2$, $V_{l_1} = \{\{1,2,3,4,5\},\{6,7,8\},\{10,11,9,12\}\}$, $V_{l_1,w_2} = \{6,7,8\}$, and $V_{l_2} = \{\{13,14,15\}\}$. The number of w-partitions for V_{l_1} is $m_1 = 3$, and $m_2 = 1$ for V_{l_2}. The number of connected components in l-partition V_{l_i} is shown with $comp(V_{l_i})$. For example, $comp(V_{l_1,w_1})$ is 2, $comp(V_{l_1,w_2})$ is 1, etc.

Constraints. The *space-partition constraint* ensures that threads executing iterations in different w-partitions need not synchronize amongst each other. The name of this constraint comes from affine partitioning [114], where the goal of the constraint is the same; however, the constraint definition is different here since the input is a DAG. If $E(V_{l_i,w_p}, V_{l_i,w_q})$ is the set of cut edges between two partitions V_{l_i,w_p} and V_{l_i,w_q}, the space-partition constraint is:

$$\forall 1 \leq i \leq n \wedge (1 \leq p,q \leq m_i), \quad E(V_{l_i,w_p}, V_{l_i,w_q}) = \emptyset \tag{4.1}$$

The w-partitions within each V_{l_j} must have no edges in common, which is the constraint expressed in Equation (4.1).

The *load balance constraint* ensures that the w-partitions within V_{l_i} are balanced up to a threshold. Assuming $\epsilon \in \mathbb{R}$ with $\epsilon \geq 0$ is a given input threshold for determining the maximum imbalance, the load balance constraint is:

$$\begin{aligned} \forall i, 1 \leq i \leq n \wedge comp(V_{l_i}) > 1 \wedge \forall 1 \leq p \in m_i, \\ cost(V_{l_i,w_p}) \leq (1+\epsilon)\lceil cost(V_{l_i})/m_i \rceil \end{aligned} \tag{4.2}$$

where $cost(V_{l_i,w_p}) = \sum_{v \in V_{l_i,w_p}} d(v)$ and $cost(V_{l_i}) = \sum_{p \in 1..m_i} cost(V_{l_i,w_p})$. As shown in Equation 4.2, the load balance constraint does not apply to an l-partition with only a single w-partition, because creating load balance for one component is not feasible. The constraint ensures that the cost of executing an l-partition V_{l_i} is uniformly distributed to w-partitions V_{l_i,w_p} so the maximum difference is less than 2ϵ.

Objective. The objective function for LBC is to reduce the critical path of the partitioned DAG, also known as quotient graph Q_G, as well as the communication cost between the partitions. Q_G is the DAG induced by the partitioning V_{l_j,w_i}, where each vertex in Q_G is a partition and edges E_q exist only if an edge exists such that the two endpoints are in separate partitions. The critical path minimization objective is to minimize P_{Q_G}. The communication cost objective is to minimize $\sum_{e \in E_q} c(e)$, where c is the cost associated with each edge of Q_G. Since no edges exist between

w-partitions, this objective minimizes the edge costs between l-partitions.

4.3.2 LBC Algorithm

As shown in Algorithm 4.1, the inputs to LBC are a DAG annotated with a cost model for both vertices and edges, the number of requested w-partitions, an architecture-related threshold, and tuning parameters *win* and *agg*. Section 4.3.3 illustrates a cost model used in LBC. Since optimizing for both l-partitions and w-partitions is complex, our algorithm uses heuristics for speed and simplicity. A major simplification is to separate the two kinds of partitioning so that the algorithm, shown in Algorithm 2, proceeds in three stages: (1) l-partitioning, (2) w-partitioning and, optionally, (3) reordering.

l-partitioning. This step finds l as defined in Section 4.3.1. The algorithm begins by finding the first partition, which contains the source nodes of the DAG; note that the upper and lower bounds for each partition represent the range of levels (the distance from the source nodes) for the vertices in the partition. In line 7, the algorithm finds the largest level (closest to the sink node of the DAG) containing enough disjoint sub-DAGs to result in approximately k w-partitions. Then, in lines 9–16 the algorithm searches through adjacent candidates up to *win* levels away for where to cut the partition, by finding the one that results in the most load-balanced w-partitions (see Section 4.3.3). Once the first l-partition is set, the loop in line 17 groups the remaining levels into l-partitions with *agg* levels per partition. Tuning parameters *win* and *agg* denote the search window for a load-balanced cut and coarseness of the remaining levels respectively. Finally, the algorithm builds the last partition, containing the sink node, in line 20.

w-partitioning. In this step, each l-partition, which is a collection of sub-DAGs with different costs, is divided into w-partitions such that the cost of each partition is balanced. To find the sub-DAGs, we do a sequence of depth-first searches from all source nodes in the l-partition. The sub-DAGs that intersect are merged. We then use a variant of the first-fit decreasing bin packing approach [98, 31] to find w-partitions with near-equal overall cost. Lines 21–26 in Algorithm 2 produce w-partitions of size k if there are enough components; otherwise, the number of bins is set to $comp(G_g)/2$. Once the balanced components are found, we use a modified breadth-first search (BFS) to store the nodes of w-partitions in a precedence order. The modified BFS algorithm starts from the source nodes of a w-partition and places the nodes in a queue. Every node that is removed from the queue is placed in the final H-Level set and then the incoming degree of its adjacent nodes is decremented. The algorithm ends when the queue is empty.

Algorithm 2: Load-Balanced Level Coarsening

Input : G, d, c k, $thresh$, win, agg
Output : V_{l_j,w_i} , l_j

/* For small DAG, use a single partition */
if $G \leq thresh$ **then**
  $V_{l_0} = G$
  l_0.lb = 0 l_0.ub = G.P
  return $\{V,l\}$
end
/* l-partitioning, starting from source nodes */
l_0.lb = 0
/* Find closest level to the sink node with enough sub-DAG */
$l_{initCut}$ = max($\{l|comp(G_{0:l}) \geq k\}$)
ϵ=∞
/* Explore cuts to find good load balance */
for i=$l_{initCut}$; i> $l_{initCut}$-win; i-=1 **do**
  CurCost(:) = BinPack($G_{0:i}$,d,k)
  maximalDiff = max(CurCost) - min(CurCost)
  if $maximalDiff < \epsilon$ **then**
    ϵ = maximalDiff
    l_0.ub = i
  end
end
/* Group rest of levels into l-partitions */
for i=$l_0.ub$; $i < G.P - agg$; i+=agg **do**
  l.append($[i, i + agg]$)
end
/* Final partition includes the sink node */
l.append($[l_n$.ub,G.P$]$)
/* w-partitioning */
for $g \in l$ **do**
  if $comp(G_g) > 1$ **then**
    parts = $comp(G_g) > k$? k : $comp(G_g)/2$
    V_g= BinPack(G_g,d,parts)
  end
end
/* Reorder w-partitions */
for i=n; i> 0; i-=1 **do**
  for j=0; j< m_i; j+=1 **do**
    $Q = \{\exists q \in child(V_{l_i,w_j})|c(e_{qV_{l_i,w_j}})$ is max $\}$
    swap(V_{l_{i+1},w_Q},V_{l_{i+1},w_j})
  end
end
return $\{V,l\}$

Because our w-partitioning algorithm merges sub-DAGs that intersect, it is possible that fewer than k components are found due to the intersection. However, we have not encountered this case in practice, and in such cases it is possible to modify the algorithm to perform w-partitioning for multiple candidate l-partitionings to find one where the most subcomponents exist.

Reordering. Optionally, the w-partitions in each l-partition can be reordered to further enhance locality. This phase reorders the computation within each w-partition

to optimize the communication cost objective. The goal is to ensure that a w-partition V_{l_j,w_i} in l-partition j that synchronizes with w-partition V_{l_{j+1},w_k} can be moved so that both w-partitions are assigned to the same thread; as a result, the data will remain local to the thread. In lines 27–32 of Algorithm 2, the LBC algorithm checks adjacent l-partitions and ensures that w-partitions with the highest communication cost are aligned vertically. During execution, w-partitions with the same ID will be assigned to the same processor, ensuring that inter-thread communication between l-partitions is minimal.

4.3.3 Cost Model & Windowing Heuristic

Statically scheduling the DAG for parallelism requires estimating the cost of each node in the DAG accurately, to ensure a high degree of parallelism and good load balance. The LBC algorithm implements two heuristics for two different parts of the algorithm that make this possible to do efficiently: a simple cost model that does not require machine-specific empirical performance measurements, and a heuristic for searching only among a small number of possible partitionings.

Existing approaches for static scheduling of sparse factorization algorithms such as that used in PaStiX [86] rely on accurate cost estimates for each BLAS operation to find load balanced partitions; PaStiX uses empirically-measured runtimes for each BLAS kernel. In contrast, the H-Level inspector uses a simple *proportional cost model* to find an efficient partitioning of the DAG. Motivated by the fact that sparse matrix computations are generally memory bandwidth-bound, this model uses the number of *participating nonzeros* in each node of the DAG as a proxy for the cost of execution.

Definition. The participating nonzeros for a node N_i in the DAG is the total number of nonzeros touched in order to complete the computation of N_i. For example, for Cholesky factorization, the participating nonzeros for a node are the nonzeros in the column block represented by N_i, plus the nonzeros touched when eliminating the block. This can be computed exactly during symbolic factorization or be approximated with the sum of the column counts for every column such that the rows corresponding to N_i have a nonzero, which can be derived in near-linear time in the size of the matrix [37]. We use a similar metric for computing edge cost, which is the number of nonzeros that must be communicated.

The proportional cost model need not be as exact as the kinds of cost models used in PaStiX, due to the much coarser granularity of scheduling in ParSy. However, any model used for static scheduling, even for coarse-grained tasks, must be accurate enough to use as a proxy for performance. This simple cost is sufficient to capture the real behavior of our static partitioning scheme. Figure 4.4 shows the actual maximal

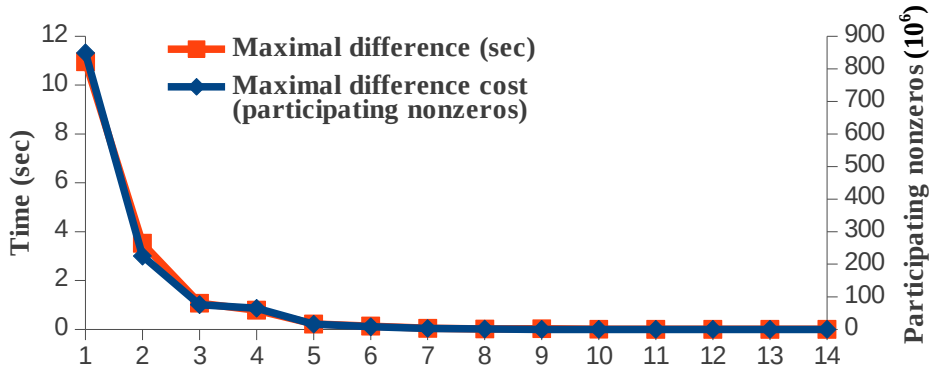


Figure 4.4: The maximal difference in time matches the maximal difference in participating nonzeros. Matrix *Flan_1565* is used as an example; other matrices exhibit similar behavior.

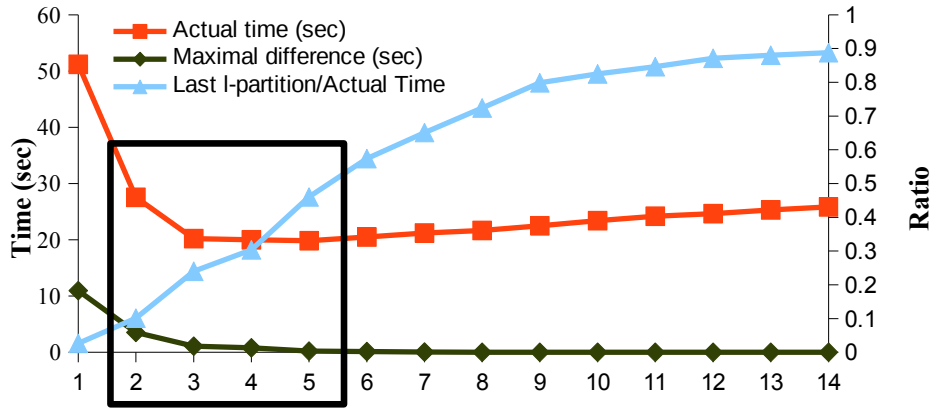


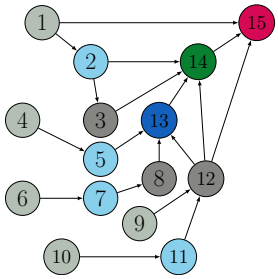
Figure 4.5: The effect of l-partitioning on the performance and load balancing of Cholesky for *Flan_1565* starting from the sink node (shown with 1) to close to the source nodes (shown with 14). The dark rectangle shows the search window from the initial point which is point 2. The line (1) in red shows the actual total runtime using each edge cut, (2) in dark green shows the maximal difference, and (3) in blue shows the percentage of actual time spent on the closest-to-sink l-partition.

difference in time versus the estimated maximal difference in cost for an example matrix based on participating nonzeros for l-partitions constructed at different levels, with the left side being cuts closest to the sink node. The cost closely matches the observed difference in time measured using cycle counters. Unlike other static partitioning schemes, the cost model used by ParSy is simple and requires no empirical measurement, while effectively estimating performance for candidate partitions.

Given this cost metric, the second heuristic tries to find the partitioning with minimal load imbalance without searching through a large number of candidates. This *windowed search* heuristic examines a small number of candidates in the neighborhood of the first l-partition containing enough sub-DAGs for parallel execution. For implementations, we use a window size (that is, the number of additional candidates to search over) of three. Figure 4.5 shows the effect of the local search. The first

l-partition with enough sub-DAGs is at point 2, but the windowing heuristic chooses a cut at point 5, which has the best load balance among candidates. As illustrated by the blue line in Figure 4.5, choosing cuts closer to the source nodes results in less work that can be done in parallel, since the l-partitions closer to the sink node cannot usually be divided into enough w-partitions for best parallel performance.

4.4 Other Sparse Matrix Methods



(a) DAG of dependencies

```
x=b; // copy RHS to x
HLevel:
for (int i=0; i<blockNo; ++i){
 b1 = block2col[i];
 b2 = block2col[i+1];
 x(b1:b2)=TRSM(L(b1:b2,b1:b2),x(b1:b2));
 tempX=GEMV(L(b2:blockNo,b1:b2),x(b1:b2));
 for(row index j in column i,k=0){
 Atomic:
 x(Li(j)) -= tempX(k++) ;}
 }}
```

(b) Serial blocked code

```
x=b;
for(every l-partition i){
#pragma omp parallel for private(tempX){
 for(every w-partition j){
  for(every v ∈ HLevelSet[i][j]){
   i = v;
   b1 = block2col[i];
   b2 = block2col[i+1];
   x(b1:b2)=TRSM(L(b1:b2,b1:b2),x(b1:b2));
   tempX=GEMV(L(b2:blockNo,b1:b2),x(b1:b2));
   for(row index j in column i,k=0){
#pragma omp atomic
    x(Li(j)) -= tempX(k++) ;}
   }}}}
```

(c) Transformed with H-level

Figure 4.6: H-Level transformation for sparse triangular solve. Figure 4.6a shows an example DAG representing the dependencies for sparse triangular solve. (b) The blocked forward substitution algorithm with compressed column format that is annotated with `HLevel` and `Atomic`. (c) Code after H-Level transformation. Gray lines in the code are not affected by the transformation.

The data dependence graphs and H-level inspection strategy in ParSy can be used for a large class of sparse matrix computations. For example, for kernels such as LU, QR, and orthogonal factorizations [116], which introduce fill-in during computation, the input DAG to ParSy is the assembly tree that captures the dependencies in the computation, including those that come from fill-ins. For kernels with no fill-in such as ILU(0), IChol(0), and triangular solve, the input is the matrix DAG. This section describes how ParSy works for sparse triangular solve, where computations are more regular than Cholesky.

Triangular Solve. This kernel solves the linear equation $Lx = b$ for x where L is a lower triangular matrix and b is the right-hand side (RHS) vector. Figure 4.6 shows two different implementations of sparse lower triangular solve for a matrix in column storage format and dense RHS. A serial implementation of the algorithm is shown in Figure 4.6b. Figure 4.6a shows the DAG of dependencies for the column-blocked version of matrix L. ParSy's H-Level inspector uses the DAG of L and builds the H-Level set which is an input for the code in Figure 4.6c. The H-Level set corresponding to the DAG in Figure 4.6a is shown in Figure 4.1b. Since the iterations in the sparse triangular solve are more regular compared to Cholesky [16], the benefits of creating

Table 4.1: Test matrices, sorted in order of decreasing parallelism. *nnz* is the number of nonzeros in L.

ID	Name	Rank (10^3)	nnz (10^6)	Parallelism (METIS)	Parallelism (SCOTCH)
1	G3_circuit	1585	127.3	16284	12154
2	ecology2	1000	54.3	11444	7454
3	thermal2	1228	71.9	10618	7087
4	apache2	715.2	164.7	10216	4427
5	StocF_1465	1465.1	1245	7755	6003
6	Hook_1498	1498	1783.8	7651	6032
7	tmt_sym	726.8	41.9	6371	4233
8	PFlow_742	742.8	598	5390	4796
9	af_shell10	1508	394.3	4900	3752
10	parabolic_fem	525.9	35	4712	3488
11	Flan_1565	1564.8	1715.9	3725	3271
12	audikw_1	943.7	1473.1	2438	2203
13	bone010	986.8	1210.1	2332	2020
14	thermomech_dM	204.3	9.7	2310	1480
15	Emilia_923	923.1	1992	2277	1927
16	Fault_639	638.8	1275.4	1595	1493
17	bmwcra_1	148.8	79.4	497	402
18	nd24k	72	435.9	48	48
19	nd12k	36	161.9	29	28

an H-Level set using ParSy are mainly in reducing synchronizations and increasing locality from level coarsening.

4.5 Experimental Results

We compare the performance of ParSy-generated code with PaStiX [86], MKL Pardiso [158], and Pardiso [158], which are specialized libraries for matrix factorization. PaStiX uses the same left-looking supernodal algorithm as ParSy and also uses a static scheduling heuristic. MKL Pardiso and Pardiso use the left-right looking supernodal variant of Cholesky and uses hybrid static/dynamic scheduling. MKL also provides optimized implementations for sparse triangular solve in compressed row, compressed column, and blocked compressed row formats. Thus, Cholesky results are compared with both PaStiX and MKL Pardiso while results for triangular solve are compared to MKL's best performing implementation amongst the three data structures. For triangular solve, we use the factorized lower-triangular matrix L that is the result of running Cholesky on each test matrix. We also parallelize each sparse kernel with the level set used in wavefront techniques [187] and call this implementation *level set*. The performance of the level set implementation is used as a baseline.

For the comparison, we use the set of symmetric positive definite matrices listed in

Table 4.1. The matrices are from [41] and belong to different domains with real number values in double precision. The testbed architectures are listed in Table 2.1. All ParSy-generated code is compiled with GCC v.5.4.0 using the `-O3` option. We report the median of 5 executions for each experiment. The PaStiX, MKL Pardiso, and Pardiso libraries are installed and executed using the recommended default configuration. For Cholesky, the default ordering method for PaStiX is Scotch [139] and for MKL Pardiso and Pardiso is Metis [103]. We use Metis ordering in ParSy for comparison to MKL Pardiso and Pardiso, and use Scotch ordering when comparing to PaStiX; this removes the effect of ordering and allows for a fair comparison. For triangular solve, we do not show the effect of reordering since reordering would possibly change the pattern of the matrix to something other than a triangular pattern. Unless otherwise stated, we include only numeric factorization time and do not include time for symbolic factorization.

Cholesky Performance. Figure 4.7 shows the performance of ParSy-generated code compared to MKL Pardiso, PaStiX, and the level set implementation. The ParSy-generated code is faster than MKL Pardiso by up to 2.7×, 1.7×, and 2.8× and is faster than PaStiX by up to 1.7×, 1.8×, and 3.1× on Haswell-E, Haswll-EP, and Skylake respectively. The speedup of ParSy over Pardiso follows the same trend as its speedup over MKL Pardiso as shown for Hasewell-E in Figure 4.7.

One of the main objectives of ParSy's inspector is to improve locality in sparse codes. Figure 4.8 shows the relationship between the performance of ParSy and MKL Pardiso to their memory accesses on the Haswell-E.The average memory access latency [85] is a measure for locality and is obtained by gathering the TLB, L1 cache, and last level cache (LLC) accesses and misses using the *perf* profiler. The Haswell-E specification parameters are obtained from [85]. Figure 4.8 demonstrates a correlation between the performance of the ParSy-generated code and the average memory access cost. The coefficient of determination or R^2 is 0.65, showing good correlation between speed-up and memory access latency. For matrices where ParSy provides better speedups, locality has been improved more. Data in Figure 4.8 shows the original measurements for the 5 runs and not the medians.

Figure 4.9 compares the ratio of wait time to CPU time in ParSy and MKL Pardiso on Haswell-E, measured using Intel's VTune Amplifier. Wait time [205] is the time that a software thread is stalled due to APIs that block or cause synchronization. CPU time [205] is the time that the CPU takes to execute numerical factorization. Because it uses dynamic scheduling, MKL Pardiso is more load balanced and thus has a nearly zero wait time for all matrices, averaging 99% CPU utilization. ParSy, however, prioritizes locality over load balance. ParSy improves locality as shown in

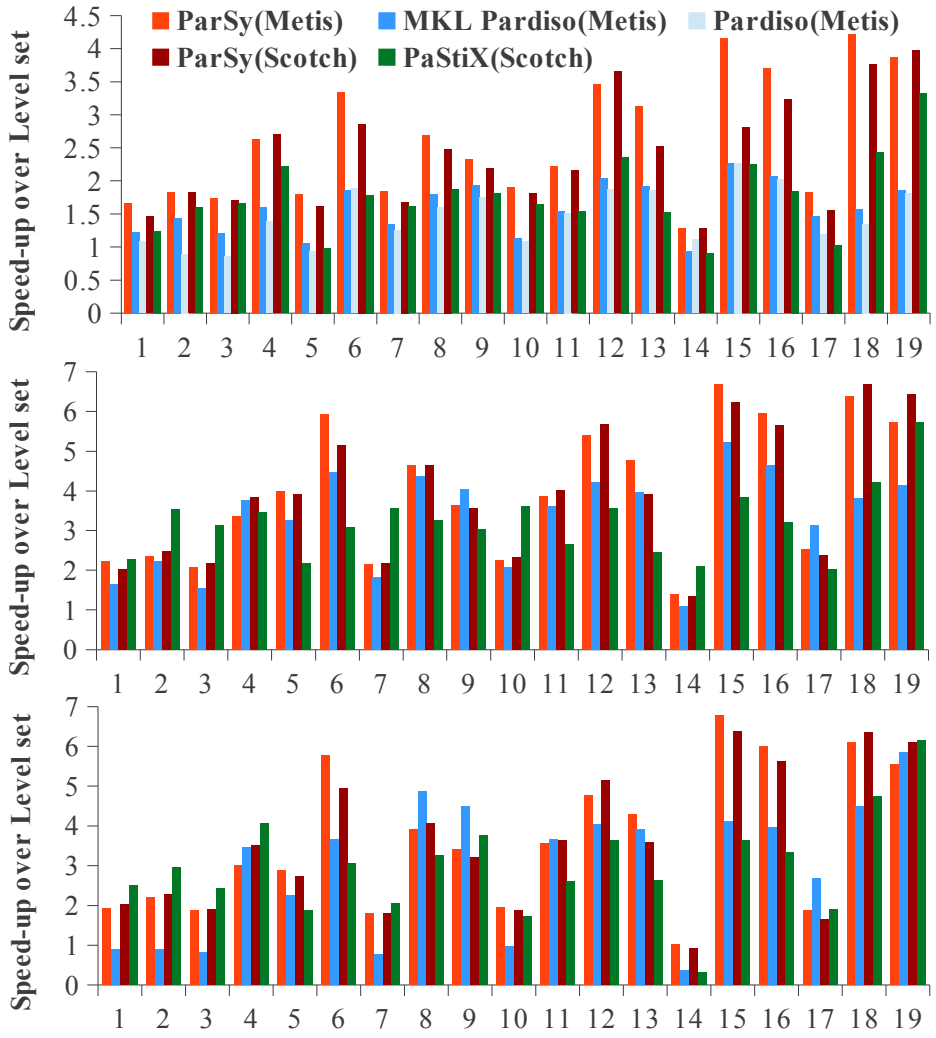


Figure 4.7: ParSy's (numeric) performance for Cholesky compared to MKL Pardiso (numeric) and PaStiX (numeric) on Haswell-E (top), Haswell-EP (middle), and Skylake (bottom). All times are normalized over the level set numeric time.

Figure 4.8 and also utilizes the CPU cores fairly efficiently with an average of 95% CPU utilization (a ratio of 0.05) as shown in Figure 4.9. Compared to MKL Pardiso, ParSy provides a better trade-off between locality and load balance which leads to the better performance results for ParSy shown in Figure 4.7.

To analyze the performance of ParSy we provide the average parallelism metric, shown with *Parallelism* in Table 4.1, which is related to the sparsity of the matrix. Parallelism is obtained by dividing the number of nodes in the DAG by its critical path and is an approximate indicator of available parallelism. The analysis based on parallelism is provided for both Metis and Scotch orderings. The performance of ParSy is shown with two different orderings. Figure 4.7 shows how the ParSy-generated code improves the performance of matrices with different sparsity patterns. The Skylake

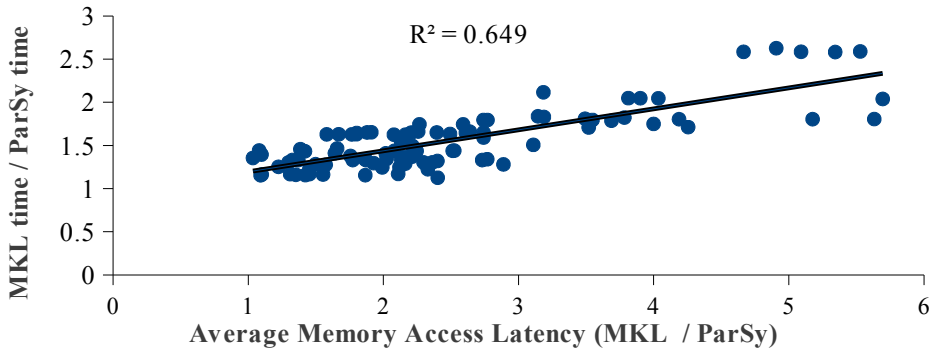


Figure 4.8: Speed up and locality relation on Haswell-E. Average memory access latency is the average cost of accessing memory in ParSy and MKL Pardiso. The relation between speed-up and the memory access ratio is approximated with a line. The coefficient of determination or R^2 of the fitted line is 0.65.

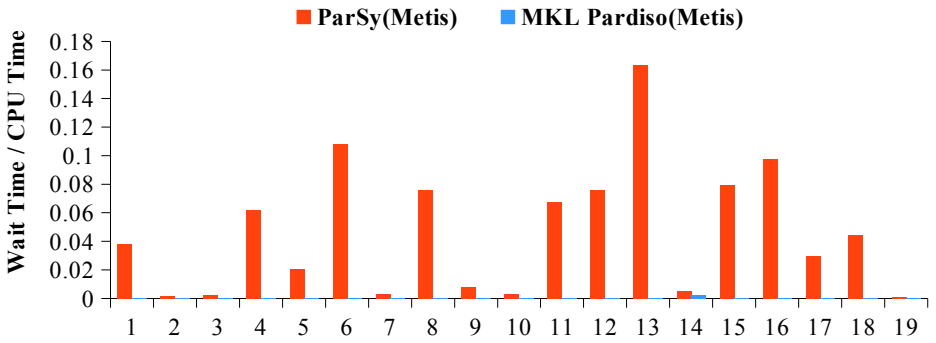


Figure 4.9: Wait time to total runtime of Cholesky's numerical factorization in ParSy and MKL Pardiso on Haswell-E.

processor has a larger number of cores compared to the other architectures; thus, we expect matrices with more parallelism to perform better with ParSy on this architecture; matrices 1, 2, and 3 which achieve high speed-ups in ParSy compared to MKL Pardiso have the most parallelism while matrices 17 and 19 with the least parallelism do not perform as well as the other matrices.

A fill-in reducing ordering method such as Metis or Scotch determines the number of nonzeros in L and affects the structure of the assembly tree. For fair comparison with libraries and to show ordering effect on ParSy, the performance of ParSy with Metis and Scotch ordering is shown in Figure 4.7. As shown, ParSy is faster than the library using the same ordering; also, ParSy performs well with both orderings. Library approaches are optimized for a specific ordering and do not perform well when the ordering is different from their default. For example, PaStiX with Metis ordering is on average 2.2× slower than PaStiX with Scotch ordering and MKL Pardiso with Scotch is on average 7.9× slower than MKL Pardiso with Metis.

Triangular Solve Performance. Figure 4.10 compares the performance of triangular solve in ParSy to MKL and wavefront parallelism. The average speed-up of

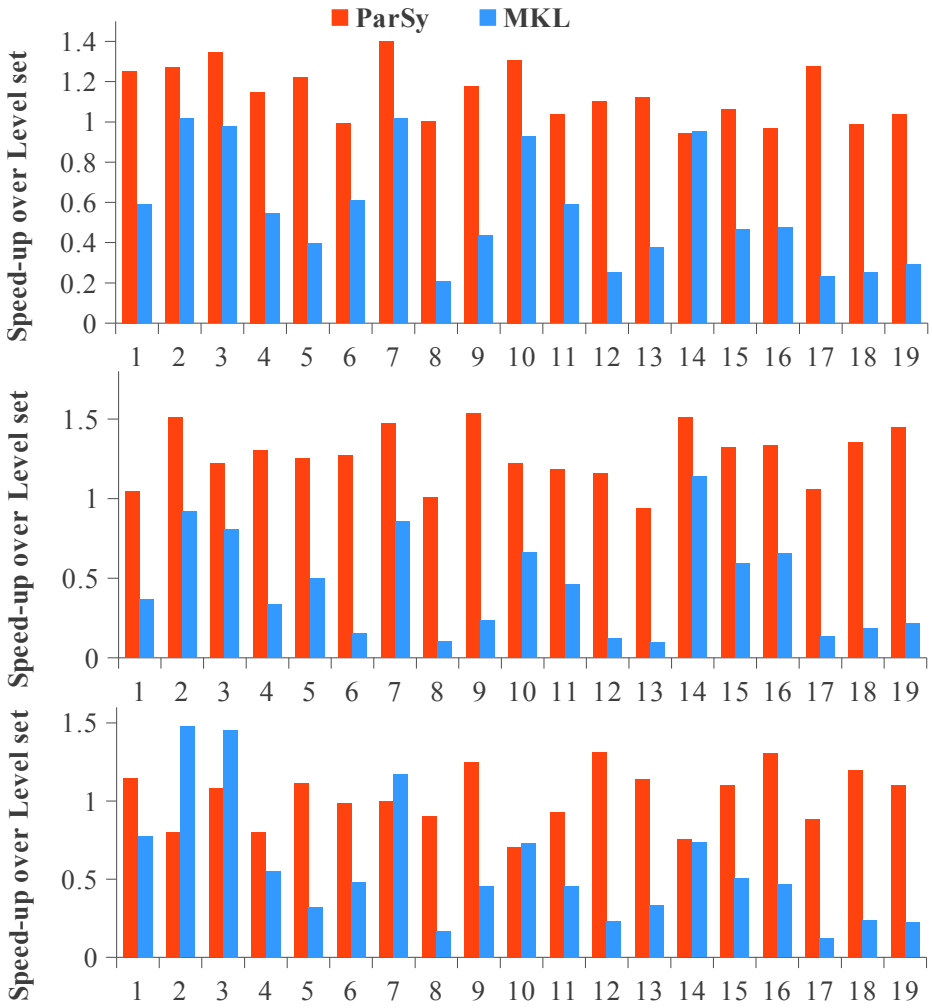


Figure 4.10: The performance of ParSy (numeric) for triangular solve compared to MKL (numeric) on Haswell-E (top), Haswell-EP (middle), and Skylake (bottom) processors. All times are normalized over the level set numeric time.

ParSy-generated code compared to the level set implementation is 1.2×, 1.3×, 1 .0× on Haswell-E, Haswell-EP, and Skylake respectively. The speed-up for triangular solve is relatively smaller than speed-ups for Cholesky. This may be due to two reasons: (1) the triangular solve is more regular, and thus the level set implementation does not create much load imbalance; (2) the kernel has less data reuse compared to Cholesky which reduces the effects of optimizing for locality. However, ParSy is faster than the highly-tuned MKL library on average by 2.6×, 4.7×, and 2.8× on Haswell-E, Haswell-EP, and Skylake respectively.

In order to test our algorithm on non-chordal DAGs, we take the matrices in Table 3.3 and modify them to include only the non-zeros in the lower triangular part of each matrix; we then run triangular solve on this synthetic lower triangular matrix. Unlike the L factors from matrix factorization, these lower triangular matrices

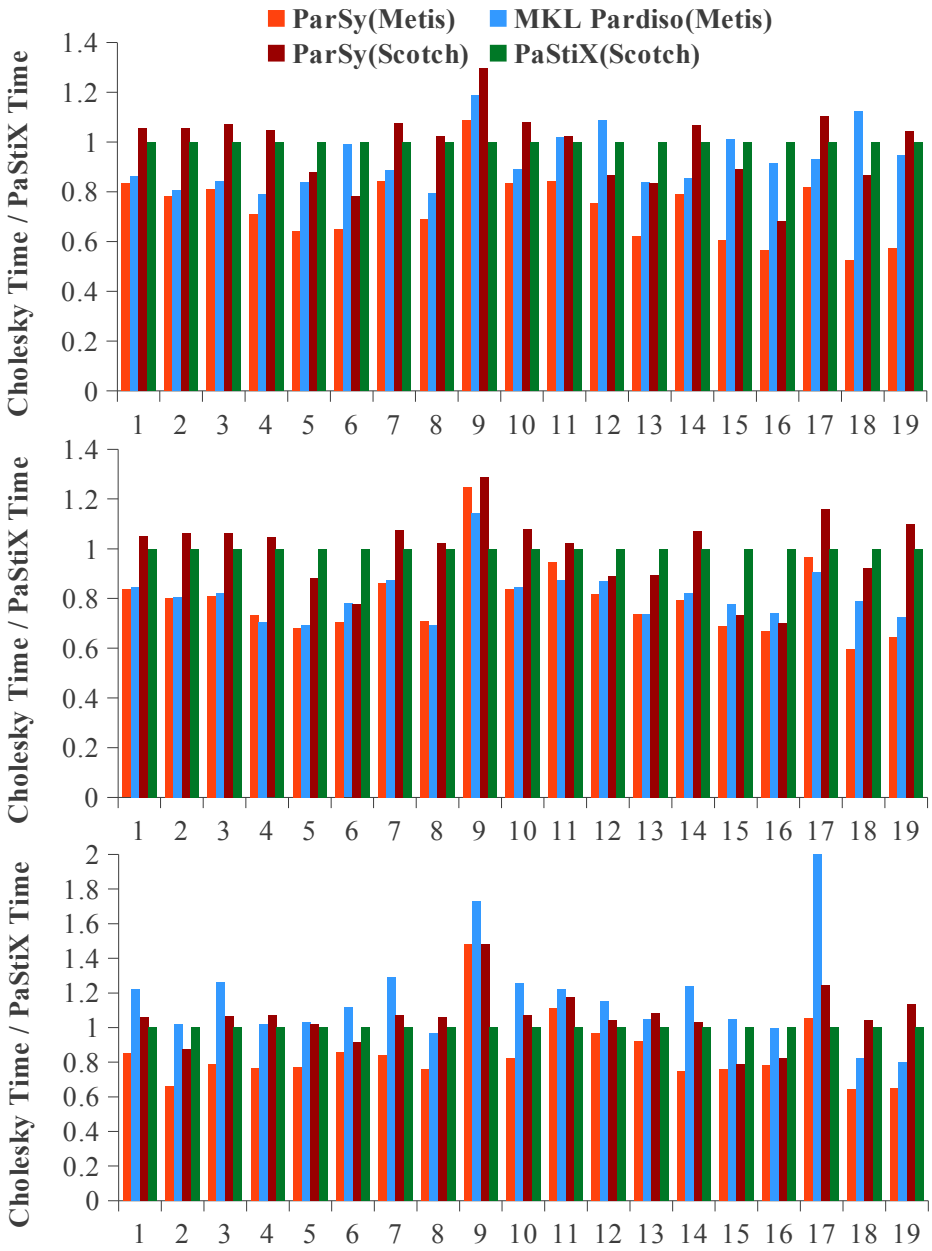


Figure 4.11: Symbolic + numeric time for ParSy-generated code, MKL Pardiso, and PaStiX for Cholesky on Haswell-E (top), Haswell-EP (middle), and Skylake (bottom). All times are normalized to PaStiX's accumulated symbolic + numeric time.

are not chordal. Figure 4.12 compares the performance of ParSy-generated code against the MKL library for the lower triangular part of matrices in Table 3.3. All matrices are first reordered with the Metis ordering method. ParSy code is faster than MKL on average by 1.6×, 2.3×, and 7.0× for Haswell-E (top), Haswell-EP (middle), and Skylake processors respectively. We observe that the heuristic approach used for finding sufficient w-partitions finds enough independent components for LBC to produce a load balanced partitioning. The number of connected components is on

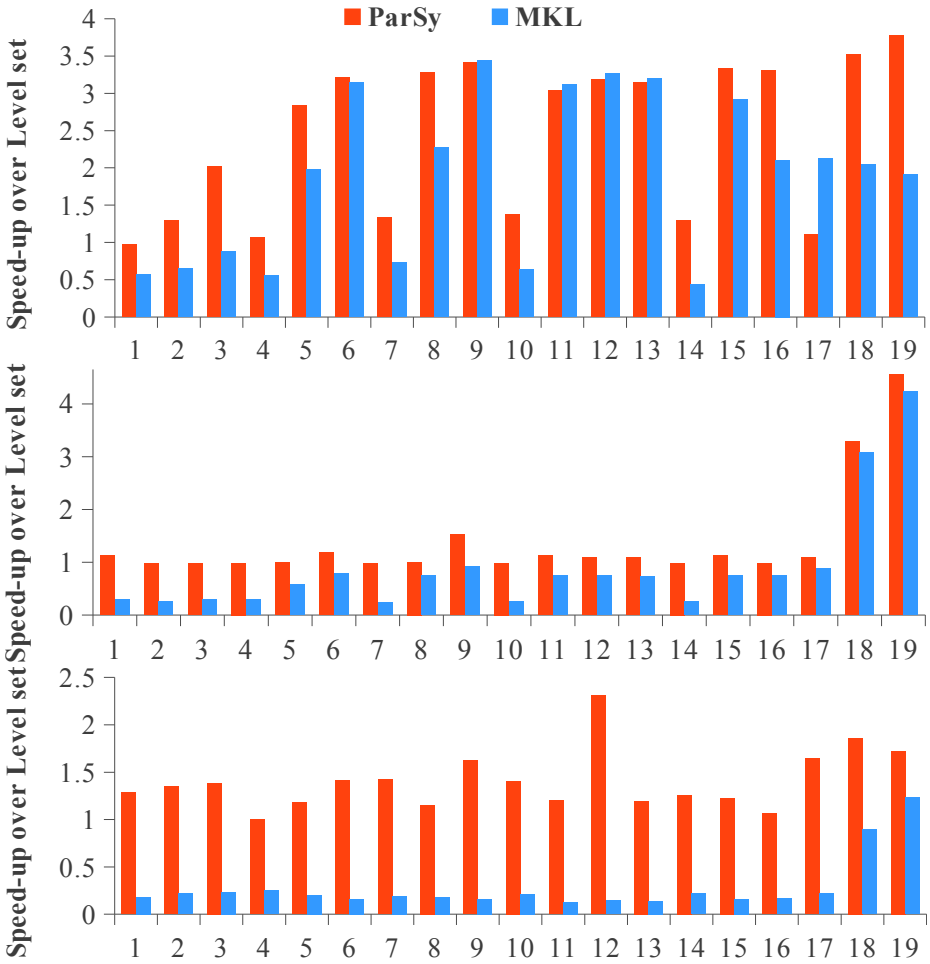


Figure 4.12: The performance of ParSy (numeric) for triangular solve on non-chordal DAGs compared to MKL (numeric) on Haswell-E (top), Haswell-EP (middle), and Skylake (bottom) processors. All times are normalized over the level set numeric time.

average 1019× the target k number of w-partitions for these matrices with non-chordal DAGs.

Inspection Overhead. The H-Level inspection is performed at compile time in ParSy and the generated code only manipulates numerical values. ParSy's accumulated time includes compile-time inspection, code generation time, and numeric factorization time. As demonstrated in Figure 4.11 , the accumulated time of ParSy is 1.3× and 1.0× faster than MKL Pardiso and PaStiX respectively, on average across all architectures. Figure 4.13 shows the accumulated time of ParSy-generated code for triangular solve is in average 4.0× and 3.4× faster than the MKL accumulated time on Haswell-E and Skylake respectively. The accumulated times for Haswell-EP follows a similar pattern to Haswell-E.

Scalability Analysis. The average speed-up for ParSy is 4×, 6.6×, and 6.8× compared to ParSy serial code on Haswell-E, Haswell-EP, and Skylake respectively.

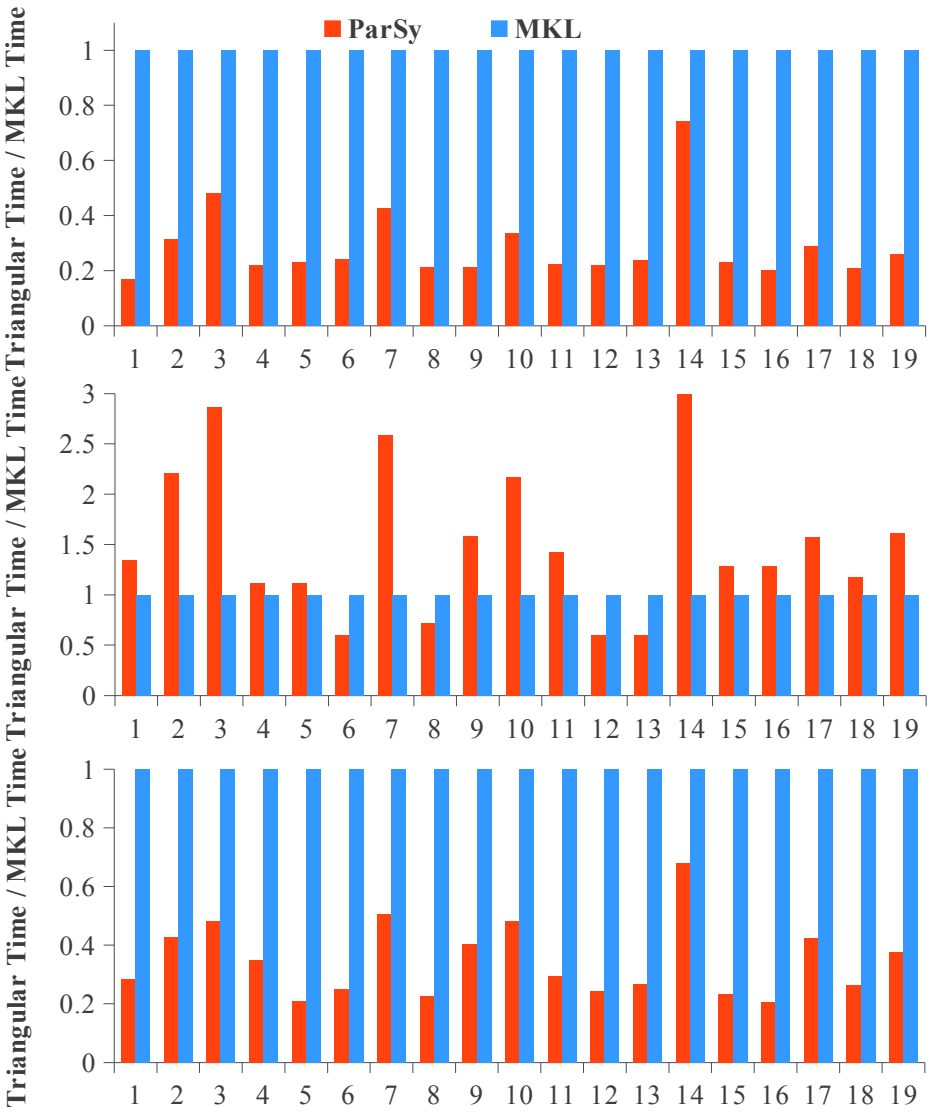


Figure 4.13: The symbolic + numeric time for ParSy-generated code and MKL for triangular solve on on Haswell-E (top), Haswell-EP (middle), and Skylake (bottom) processors. All times are normalized to MKL's accumulated symbolic + numeric time.

For MKL Pardiso and PaStiX the average speed-ups compared to their own serial codes are 3.9×, 7.8×, and 8.4× for MKL Pardiso and 4.3×, 7.4×, 7.5× for PaStiX for Haswell-E, Haswell-EP, and Skylake. These numbers demonstrate good scaling in all three implementations. However, the performance of ParSy is 1.4× faster than the two libraries across all architectures.

4.6 Related Work

Wavefront parallelism [187, 148, 204, 172, 132, 77] is one of the most common approaches inspector-executor frameworks use to parallelize sparse matrix methods.

These either employ manually-written inspectors and executors [172, 132, 77, 137, 117] or automate parts of the process by simplifying the inspector [187, 148, 204, 67]. These approaches use inspectors to obtain dependence information that is only known at runtime. The H-Level sets created in ParSy are typically coarser than level sets in wavefront parallelism, reducing the number of costly synchronizations. ParSy also improves load balance in irregular sparse codes such as Cholesky compared to wavefront approaches. The closest approach to ours that finds an efficient trade-off between locality and load balance is in [16], which extends the Pluto framework [21] with an automatic parallelization approach for transforming input affine sequential codes. However, this is limited to structured and dense kernels.

Numerous hand-optimized parallel sparse libraries exist with efficient sparse matrix kernels. These libraries differ in numerical methods they optimize and the platforms supported. Implementations in [37, 26, 42] provide sequential sparse kernels such as LU and Cholesky while parallel implementations exist in work such as SuperLU [45], MKL Pardiso [158], and PaStiX [86] for shared memory architectures, and in [45, 7] for distributed memory. Several libraries have also optimized specific sparse kernels such as triangular solve [110, 132, 193, 189, 182] and sparse matrix-vector multiply [195, 100, 124]. Sparse kernel variants differ between libraries; for example, PaStiX implements left-looking sparse Cholesky while MKL Pardiso uses a left-right looking approach [156]. ParSy optimizes left-looking Cholesky on shared memory architectures.

Parallel sparse libraries use numerical method-specific code to determine data dependencies and schedule the computation. These libraries typically inspect the symbolic information of the matrix, which is called static/symbolic analysis, and use the information for numerical manipulation with the objective of creating load-balanced tasks that can execute in parallel. Libraries such as PaStiX [86] use static analysis and static scheduling [2] while most other libraries use hybrid static/dynamic [167, 156] scheduling. Typically the DAG is partitioned during inspection with algorithms such as the subtree-to-subcube heuristic [63, 141, 102]. While dynamic scheduling can introduce overheads at runtime, static schedulers using profiling data on a specific architecture limit portability. ParSy uses the matrix structure and numerical method to compute a proportional cost that does not rely on the underlying architecture and enables compile-time scheduling of tasks.

Chapter 5

Sparse Fusion

Sympiler inspects the computation pattern of an individual computation and generates efficient codes that run efficiently on a single core and a multicore processor as discussed in Chapters 3 and 4. However, several numerical algorithms [152] and optimization methods [23, 170, 30] in scientific simulations and data analytics codes are typically composed of numerous sparse matrix computations, and there are optimization opportunities between them. For example, in iterative solvers [152] such as Krylov methods [153], sparse kernels that apply a preconditioner or update the residual are repeatedly executed inside and between iterations of the solver. Sparse kernels with loop-carried dependencies, i.e. kernels with partial parallelism, are frequently used in numerical algorithms, and the performance of scientific simulations relies heavily on efficient parallel implementations of these computations. Optimizing such kernels, especially when done separately results in synchronization overhead and load imbalance. In this chapter, sparse fusion is introduced that creates an efficient schedule and fused code for when a sparse kernel with loop-carried dependencies is combined with another sparse kernel. Sparse fusion uses an inspector to apply a novel Multi-Sparse DAG Partitioning (MSP) method on the DAGs of the two input sparse kernels.

Sparse kernels that exhibit partial parallelism often have multiple wavefronts of parallel computation where a synchronization is required for each wavefront, i.e. wavefront parallelism [187, 77]. The amount of parallelism varies per wavefront and often tapers off towards the end of the computation, which results in load imbalance. Figure 5.1 shows with dark lines the nonuniform parallelism for the sparse incomplete Cholesky (SpIC0) and the sparse triangular solve (SpTRSV) kernels when SpTRSV executes after SpIC0 completes. Separately optimizing such kernels exacerbates this problem by adding even more synchronization. Also, opportunities for data reuse between two sparse computations might not be realized when sparse kernels are op-

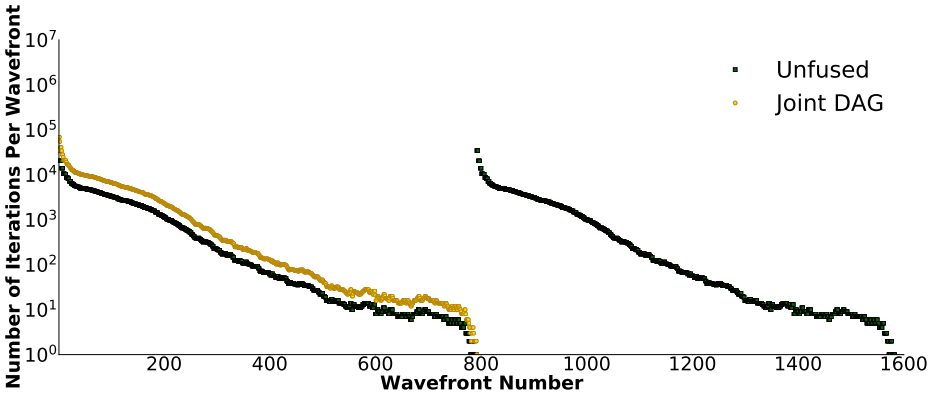


Figure 5.1: The nonuniform parallelism in the DAGs of sparse incomplete Cholesky and triangular solver (annotated with unfused) and for the joint DAG of the two kernels results in load imbalance. Higher value in the y-axis shows high parallelism in a given wavefront. Wavefront numbers in the x-axis are numbered based on their order of execution.

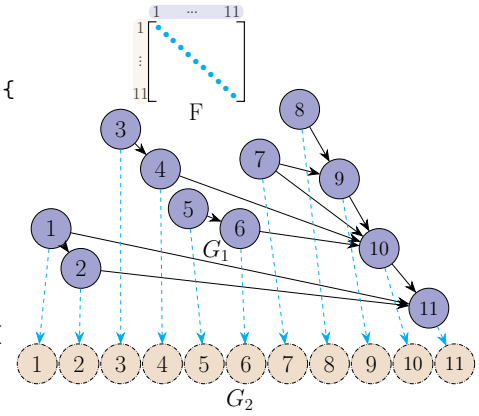
timized separately.

Instead of scheduling iterations of sparse kernels separately, they can be scheduled jointly. Wavefront parallelism can be applied to the joint DAG of two sparse computations. A data flow directed acyclic graph (DAG) describes dependencies between iterations of a kernel [29, 174, 86]. A joint DAG includes all of the dependencies between iterations within and across kernels. The joint DAG of sparse kernels with partial parallelism with the DAG of another sparse kernel provides slightly more parallelism per wavefront without increasing the number of wavefronts. The yellow line in Figure 5.1 shows how scheduling the joint DAG of SpIC0 and SpTRSV provides more parallelism per wavefront and significantly reduces the number of wavefronts (synchronizations). However, the load balance issues remain, and there are still several synchronizations.

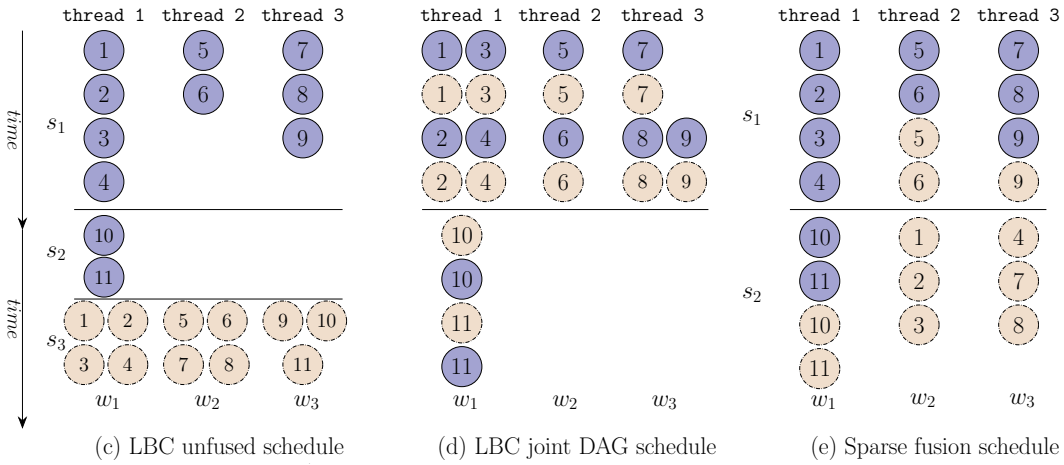
Wavefronts of the joint DAG can be aggregated to reduce the number of synchronizations. DAG partitioners such as Load-Balanced Level Coarsening (LBC) [28] and DAGP [89] apply aggregation, however, when applied to the joint DAG because they aggregate iterations from consecutive wavefronts, load imbalance might still occur. Also, by aggregating iterations from wavefronts in the joint DAG, DAG partitioning methods potentially improve the temporal locality between the two kernels but, this can disturb spatial locality within each kernel. For example, for two sparse kernels that only share a small array and operate on different sparse matrices, optimizing temporal locality between kernels will not be profitable. Finally, even when applied to the DAG of an individual kernel, DAGP and LBC are slow for large DAGs because of the overheads of coarsening [89]. This problem exacerbates when applied to the joint because the joint DAG is typically 2-4× larger than an individual kernel's

```
// Sparse Lower Triangular Solve
// (SpTRSV) Lx = b
Fuse:for(i1=0;i1<n;i1++){
 for(i2=Lp[i1];i2<Lp[i1+1]-1;i2++){
  x[i1] -= Lx[i2]*x[Li[i2]];
 }
  x[i1] /= Lx[Lp[i1+1]-1];
}
// Sparse Matrix-Vector Multiply
// (SpMV) y = Ax
Fuse:for(j1=0;j1<n;j1++){
 for (j2=Ap[j1];j2<Ap[j1+1];j2++){
 Atomic:y[Ai[j2]] += Ax[j2]*x[j1];
 }
}
```

(a) Initial AST



(b) Dependency DAGs and matrix



(c) LBC unfused schedule

(d) LBC joint DAG schedule

(e) Sparse fusion schedule

Figure 5.2: Figures 5.2c-5.2e show three different schedules for running a sparse lower triangular kernel (SpTRSV) followed by a sparse matrix-vector multiplication (SpMV) as shown in Figure 5.2b. We choose the number of processors (r) to be three. Solid purple (G_1) and dash-dotted yellow (G_2) vertices in order represent iterations of SpTRSV and SpMV and edges show the dependencies between iterations. Dashed edges in Figure 5.2b show dependencies between two kernels and correspond to the nonzero elements of matrix F. The unfused implementation schedules each DAG separately as shown in Figure 5.2c. Two different fused implementations in Figure 5.2d and 5.2e use both DAGs and dependencies between kernels to build a fused schedule.

DAG[1].

This chapter presents sparse fusion that creates an efficient schedule and fused code for when a sparse kernel with loop-carried dependencies is combined with another sparse kernel. Sparse fusion uses an inspector to apply a novel Multi-Sparse DAG Partitioning (MSP) runtime scheduling algorithm on the DAGs of the two input sparse kernels. MSP uses a vertex dispersion strategy to balance workloads in the

[1]The DAGP and LBC runtimes for different DAG sizes are shown in Section A.1.

fused schedule, uses two novel iteration packing heuristics to improve the data locality due to spatial and temporal locality of the merged computations, and uses vertex pairing strategies to aggregate iterations without joining the DAGs.

Figure 5.2 compares the schedule created by sparse fusion (sparse fusion schedule) with the schedules created by applying LBC to the individual DAGs of each sparse kernels (LBC unfused schedule) and LBC applied to the joint DAG (LBC joint DAG schedule). All approaches take the input DAGs in Figure 5.2b. Solid purple vertices are the DAG of sparse triangular solve (SpTRSV) and the dash-dotted yellow correspond to Sparse Matrix-Vector multiplication (SpMV). LBC is a DAG partitioner that partitions a DAG into a set of aggregated wavefronts called s-partitions[2] that run sequentially, each s-partition is composed of some independent w-partitions. In the LBC unfused schedule in Figure 5.2c, LBC is used to partition the SpTRSV DAG and will create two s-partitions, *i.e.* s_1 and s_2. The vertices of SpMV are scheduled to run in parallel in a separate wavefront s_3. This implementation is not load balanced because the number of partitions that can run in parallel differs for each s-partition. In the LBC joint DAG schedule, the DAGs are first joint using the dependency information between the two kernels shown with blue dotted arrows and then LBC is applied to create the two s-partitions in Figure 5.2d. These s-partitions are also not load balanced, for example s_2 only has one partition. Sparse fusion uses MSP to first partition the SpTRSV DAG and then disperses the SpMV iterations to create load-balanced s-partitions, e.g. the two s-partitions in Figure 5.2e have three closely balanced partitions.

SpTRSV solves $Lx = b$ to find x and SpMV performs $y = A * x$ where L is a sparse lower triangular matrix, A is a sparse matrix, and x, b, and y are vectors. The LBC joint DAG schedule interleaves iterations of two kernels to reuse `x`. However, this can disturb spatial locality within each kernel because the shared data between the two kernels, x, is smaller than the amount of data used within each kernel, A and L. With the help of a reuse metric, sparse fusion realizes the larger data accesses inside each kernel and hence packs iterations to improve spatial locality within each kernel.

We implement sparse fusion as an embedded domain-specific language in `C++` that takes the specifications of the sparse kernels as input, inspects the code of the two kernels, and transforms code to generate an efficient and correct parallel fused code. The primary focus of sparse fusion is to fuse two sparse kernels where at least one of the kernels has loop-carried dependence. Sparse fusion is tested on seven of the most commonly used sparse kernel combinations in scientific codes which include kernels such as sparse triangular solver, incomplete Cholesky, incomplete LU, diagonal

[2]S-partitions are more general than l-partitions previously defined in Chapter 4. An l-partition contains all vertices within a wavefront range while a s-partition contain some or all of them.

```
#include "def.h"
void main(){
 Int n;
 Int r(MAX_THREADS);
 CSR L(n,n,"./L.mtx");
 CSC A(n,n,"./A.mtx");
 Vec x(n), y(n);
 Vec b(n,"./b.mtx");
 ...
 Fuse TM(
    SpTRSV(L,b,x),
    SpMV(A,x,y)
   );
 TM.gen_c("TrsvMv.h"
     , "Driver.cpp",r);}
```

(a) Input specification

```
#include "TrsvMv.h"
#include "MSP.h"
void main(){
 L.load();A.load();b.load();
/// -------- Inspector -------- ///
 G1 = SpTRSV.intra_DAG(L);//Sec 2.2
 G2 = SpMV.intra_DAG(A);
 F = inter_DAG(A,L,b,x,y);//Sec 2.2
 reuse_ratio = compute_reuse(
   A,L,b,x,y); //Sec 2.2
 FusedSchedule = MSP(G1,G2,F,
   r,reuse_ratio); //Sec 3
/// -------- Executor -------- ///
 fused_code(L,b,A,x,y,FusedSchedule,
   reuse_ratio); /*Sec 2.3*/}
```

(b) Driver code (`driver.cpp`)

Figure 5.3: Sparse fusion's input and the driver code.

scaling, and matrix-vector multiplication. The generated code is evaluated against MKL and ParSy with average speedups of 5.1× and 1.6× respectively. Sparse fusion compared to fused implementations of LBC, DAGP, and wavefront techniques applied to the joint DAG provides on average 5.1×, 7.2× and 2.5× speedup respectively.

```
Fuse:for(I1){//loop 1
 ...
  for(In)
   x[h(I1,...,In)] = a*y[g(I1,...,In)];
 }
Fuse:for(J1){//loop 2
...
  for(Jm)
   z[h'(J1,...,Jm)] = a*x[g'(J1,...,Jm)];
 }
```

(a) Before

```
if(FusedSchedule.fusion && reuse_ratio < 1){
 for (every s-partition s){
#pragma omp parallel for
  for (every w-partition w){
   for(v ∈ FusedSchedule[s][w].L1){//loop 1
    ...
    for(In)
      x[h(v,...,In)] = a*y[g(v,...,In)];
   }
   for(v ∈ FusedSchedule[s][w].L2){//loop 2
    ...
    for(Jm)
      z[h'(v,...,Jm)] = a*x[g'(v,...,Jm)];
   }}}}
```

(b) After - separated variant

```
if(FusedSchedule.fusion && reuse_ratio >= 1){
 for (every s-partition s){
#pragma omp parallel for
  for (every w-partition w){
   for(v ∈ FusedSchedule[s][w]){
    if(v.type == L1){//loop 1
     for(In)
      x[h(v.id,...,In)] = a*y[g(v,...,In)];

    } else {//loop 2
     for(Jm)
      z[h'(v.id,...,Jm)] = a*x[g'(v,...,Jm)];
    }
   }}}}
```

(c) After - interleaved variant

Figure 5.4: The general form of the sparse fusion code transformation with its two variants, interleaved and separated. `I1...In` and `J1...Jm` represent two loop nests. `h'` and `g'` are data access functions. `FusedSchedule` contains the schedule for iterations of loops `I1`, shown with `L1` and `J1`, shown with `L2`.

5.1 Sparse Fusion

Sparse fusion is implemented as a code generator with an inspector-executor technique that can be used as a library. It takes the input specification shown in Figure 5.3a and generates the inspector and the executor in Figure 5.3b. The inspector includes the MSP algorithm and functions that generate its inputs, i.e. dependency DAGs, reuse ratio, and the dependency matrix. The executor is the fused code that is created by the fused transformation.

5.1.1 Code Generation

Sparse fusion is implemented as an embedded domain-specific language. It takes as input the specification shown in Figure 5.3a and generates the driver code in Figure 5.3b. At compile-time, the data types and kernels in Figure 5.3a are converted to an initial Abstract Syntax Tree (AST) using `TM.gen_c()` in line 14. Lines 11 and lines 12 in Figure 3a demonstrate how the user specifies the two kernels for the running example in Figure 2 as inputs to sparse fusion. The corresponding AST for the example is shown in Figure 2a.

At runtime by running the driver code in Figure 5.3b, the inspector creates a fused schedule, and the executor runs the fused schedule. The inspector first builds inputs to MSP using functions `intra_DAG`, `inter_DAG`, and `compute_reuse` in lines 6–10 in Figure 5.3b and then calls `MSP` in line 11 to generate `FusedSchedule` for `r` threads. Then the executor code, `fused_code` in line 14 in Figure 5.3b, runs in parallel using the fused schedule.

5.1.2 The Inspector in Sparse Fusion

The MSP algorithm requires kernel-specific inputs. Its inputs are the dependency matrix between kernels, the DAG of each kernel, and a reuse ratio. Sparse fusion analyzes the kernel code, available from its AST, to generate inspector components that create these inputs.

Dependency DAGs: Lines 6–7 in Figure 5.3b use an internal domain-specific library to generate the dependency DAG of each kernel. General approaches such as work by Mohammadi et al. [127, 126, 166] can also be used to generate the DAGs however, that will lead to higher inspection times compared to a domain-specific approach. For example, with domain knowledge, sparse fusion will use the L matrix as the SpTRSV DAG G_1 in Figure 5.2b. Each nonzero L_{ij} represents a dependency from iteration i to j.

Dependency Matrix F: MSP uses the dependency information between kernels to create a correct fused schedule. By running the `inter_DAG` function, sparse fusion creates this information and stores it in matrix F. To generate `inter_DAG`, sparse fusion finds dependencies between statements of the two kernels by analyzing the AST. Each nonzero $F_{i,j}$ represents a dependency from iteration j of the first loop, i.e. column j of F, to iteration i of the second loop, i.e. row i of F. In Figure 5.2a, there exists a read after write (flow) dependency between statements `x[i1]` in line 5 and `x[j1]` in line 13. As a result, sparse fusion generates the function shown in Listing 5.1. The resulting F matrix, generated at runtime, is shown in Figure 2b.

```
for(i1=0; i1<n; i1++){
 j1 = i1;
 if(A.p[j1] < A.p[j1+1] )
  F[j1].append(i1); }
```

Listing 5.1: `inter_DAG` function for the example in Figure 5.2a.

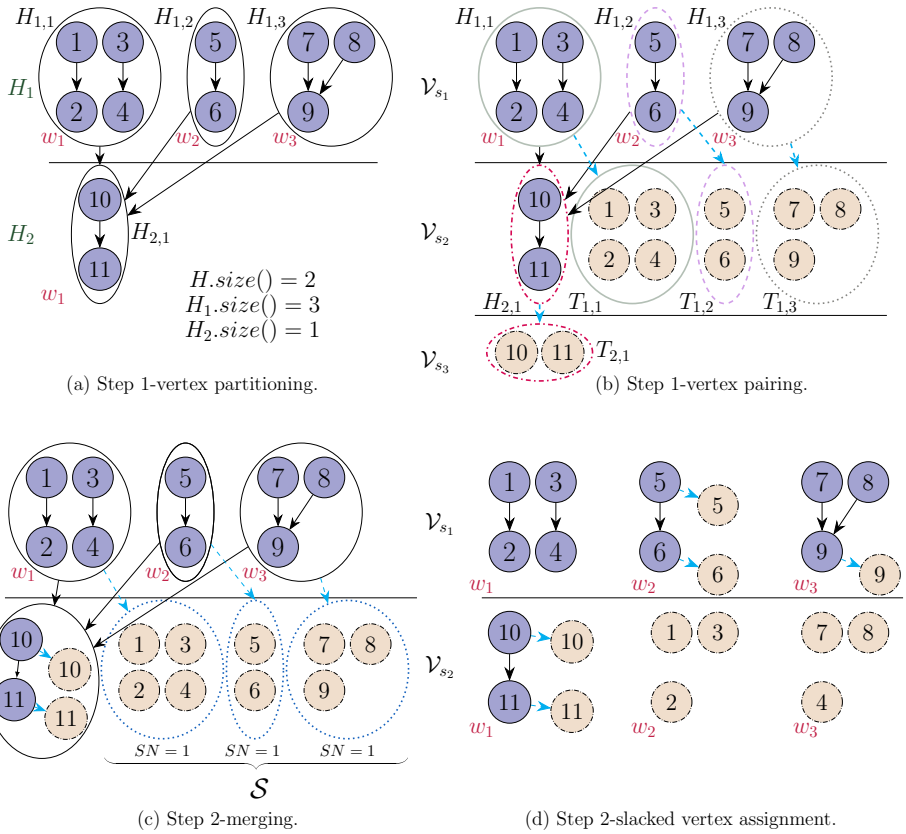
Reuse Ratio: MSP uses a reuse ratio based on the memory access patterns of the kernels to decide whether to improve locality within each kernel or between the kernels. The inspector in line 9 in Figure 5.3b computes the reuse ratio metric. The metric represents the ratio of common to total memory accesses of the two kernels, i.e. $\frac{\text{common memory access}}{max(\text{kernel1 accesses, kernel2 accesses})}$. For a reuse ratio larger than one, the number of common memory accesses between the two kernels is larger than the accesses inside a kernel. Sparse fusion estimates memory accesses using the ratio of the size of common variables over the maximum of the total size of variables amongst the kernels. For the running example, the code generated for `compute_reuse` is `2*x.n / max(A.size+x.n+y.n,L.size+x.n+b.n)`. Since `x` is smaller than L or A, the reuse ratio is less than one.

5.1.3 Fused Code

To generate the fused code, a fused transformation is applied to the initial AST at compile-time and two variants of the fused code are generated, shown in Figure 5.4. The transformation variants are *separated* and *interleaved*. The fused code uses the reuse ratio at runtime to select the correct variant for the specific input. The variable `fusion` in line 1 of Figure 5.4b and 5.4c is set to `False` if MSP determines fusion is not profitable. Figure 5.4a shows the sequential loops in the AST, which are annotated with `Fuse`, and are transformed to the separated and interleaved code variants as shown in order in Figures 5.4b and 5.4c. The separated variant is selected when the reuse ratio is smaller than one. In this variant, iterations of one of the loops run

consecutively without checking the loop type. The interleaved variant is chosen when the reuse ratio is larger than one. In this variant, iterations of both loops should run interleaved, and the variant checks the loop type per iteration as shown in lines 6 and 10 in Figure 5.4c.

5.2 Multi-Sparse DAG Partitioning



(a) Step 1-vertex partitioning.

(b) Step 1-vertex pairing.

(c) Step 2-merging.

(d) Step 2-slacked vertex assignment.

Figure 5.5: Stages of MSP for DAGs G_1 and G_2 and matrix F in the running example shown in Figure 5.2b where the reuse ratio ($reuse\_ratio$) is smaller than one and number of processors (r) is three. The first step of the algorithm selects G_1 and creates H partitioning for three processors using the LBC algorithm as shown in Figure 5.5a. Then it pairs each $H_{i,j}$ through dependencies in matrix F to create partitioning T of G_2 as shown in Figure 5.5b. The partitions with the same line pattern/color are pair partitions. In the second step, MSP merges pair partitions that cannot be dispersed such as first w-partitions of s-partitions 2 and 3 ($\mathcal{V}_{s_3,w_1}$ and $\mathcal{V}_{s_2,w_1}$) in Figure 5.5b, these are merged into $\mathcal{V}_{s_2,w_1}$ in Figure 5.5c. Slack vertices, which are denoted as $\mathcal{S}$ are shown with blue dotted circles in Figure 5.5c. Slack vertices are assigned into imbalanced w-partitions as shown in Figure 5.5d. Since the reuse ratio is smaller than one, vertices inside each partition are packed separately as shown in Figure 5.2e.

Sparse fusion uses the multi-sparse DAG partitioning (MSP) algorithm to create an efficient fused partitioning that will be used to schedule iterations of the fused code. MSP partitions vertices of the DAGs of the two input kernels to create parallel load-

balanced workloads for all cores while improving locality within each thread. This section describes the inputs, output, and three steps of the MSP algorithm using the running **example** in Figures 5.2 and 5.5.

Algorithm 3: The MSP algorithm.

Input : $G_1(V_1, E_1, c_1)$, $G_2(V_2, E_2, c_2)$, F, r, $reuse\_ratio$
Output: $\mathcal{V}$
/* (i) Vertex partitioning and partition pairing */
if $|E_2| > 0$ **then**
  $[H, k] = \text{LBC}(G_2, r).\text{list}()$, $T = \emptyset$, $\mathcal{V} = \emptyset$
  /* Backward pairing */
  for $(i = 1 : H.size())$ **do**
    for $(j = 1 : H_i.size())$ **do**
      $T_{i,j} = \text{BFS}(H_{i,j}, F, G_1)$
      $\mathcal{V}.add(T_{i,j}, H_{i,j})$
    end
  end
  if $|\mathcal{V}| > 2 \times (|V_1| + |V_2|)$ **then** $\mathcal{V}$.fusion = False, exit()
else
  $[H, k] = \text{LBC}(G_1, r).\text{list}()$, $T = \emptyset$, $\mathcal{V} = \emptyset$
  /* Forward pairing */
  for $(i = 1 : H.size())$ **do**
    for $(j = 1 : H_i.size())$ **do**
      $T_{i,j} = \text{BFS}(H_{i,j}, F^T, G_2)$
      $U_{i,j} = T_{i,j}.\text{remove\_uncontained}(F)$
      $\mathcal{V}.add(H_{i,j}, T_{i,j}, U_{i,j})$
    end
  end
end
/* (ii) Merging and slacked vertex assignment */
$\mathcal{S} = \text{slack\_info}(\mathcal{V})$
for *(every w-partition pair* $(w, w') \in \mathcal{V}.pairs$**)** **do**
  if $(SN(w) = 0) \wedge (SN(w') = 0)$ **then** $\mathcal{V}$.merge(w,w')
end
$\mathcal{V} = \mathcal{V} - \mathcal{S}$, $\epsilon = |\mathcal{V}| \times 0.001$
for $(i = 1 : \mathcal{V}.b)$ **do**
  for $(j = 1 : m_i)$ **do**
    if $max\_diff(\mathcal{V}_{s_i}, \mathcal{V}_{s_i,w_j}) > \epsilon \wedge \mathcal{S} \neq \emptyset$ **then** $\mathcal{S} = \mathcal{V}_{s_i,w_j}.\text{balance\_with\_pair}(\mathcal{S})$
    if $max\_diff(\mathcal{V}_{s_i}, \mathcal{V}_{s_i,w_j}) > \epsilon \wedge \mathcal{S} \neq \emptyset$ **then** $\mathcal{S} = \mathcal{V}_{s_i,w_j}.\text{balance\_with\_slacks}(\mathcal{S})$
  end
  if $\mathcal{S} \neq \emptyset$ **then** $\mathcal{S} = \mathcal{V}_{s_i}.\text{assign\_even}(\mathcal{S})$
end
/* (iii) Packing */
if $reuse\_ratio \geq 1$ **then** $\mathcal{V}$.interleaved_pack(F)
else $\mathcal{V}$.separated_pack()

5.2.1 Inputs and Output to MSP

The inputs to MSP (shown in Algorithm 3) are two DAGs G_1 and G_2 from in order lexicographically first and second input kernels, and the inter-DAG dependency matrix F that stores the dependencies between kernels. A DAG shown with $G_j(V_j, E_j, c)$ has a vertex set V_j and an edge set E_j and a non-negative integer weight $c(v_i)$ for each

vertex $v_i \in V_j$. The vertex v_i of G_j represents iteration i of a kernel and each edge shows a dependency between two iterations of a kernel. $c(v_i)$ is the computational load of a vertex and is defined as the total number of nonzeros touched to complete its computation. Because sparse matrix computations are generally memory bandwidth-bound, $c(v_i)$ is a good metric to evaluate load balance in the algorithm [28]. F is stored in the compressed sparse row (CSR) format and F_i is used to extract the set of vertices in G_1 that $v_i \in V_2$ depends on. Other inputs to the algorithm are the number of requested partitions r, which is set to the number of cores, and the reuse ratio discussed in section 5.1.2.

The output of MSP is a *fused partitioning* $\mathcal{V}$ that has $b \geq 1$ s-partitions, each s-partition contains up to $k > 1$ w-partitions, where $k \leq r$. MSP creates b disjoint s-partitions from vertices of both DAGs, shown with $\mathcal{V}_{s_i}$ where $\cup_{i=0}^{b} \mathcal{V}_{s_i} = V_1 \cup V_2$. Each s-partition includes vertices from a lower bound and upper bound of wavefront numbers shown with $s_i = [lb_i..ub_i)$ as well as some *slack vertices*. For each s-partition $\mathcal{V}_{s_i}$, MSP creates $m_i \leq k$ independent *w-partitions* $\mathcal{V}_{s_i,w_j}$ where $\mathcal{V}_{s_i,w_1} \cup ... \cup \mathcal{V}_{s_i,w_{m_i}} = \mathcal{V}_{s_i}$. Since w-partitions are independent, they can run in parallel.

Example. In Figure 5.2b, the SpTRSV DAG G_1, the SpMV DAG G_2, the inter-DAG dependency matrix F are inputs to MSP. Other inputs to MSP are r=3 and the *reuse_ratio*. The fused partitioning shown in Figure 5.2e has two s-partitions (b=2). The first s-partition has three w-partitions (m_1=3) shown with $\mathcal{V}_{s_1} = \{[\underline{1}, \underline{2}, \underline{3}, \underline{4}]; [\underline{5}, \underline{6}, 5, 6]; [\underline{7}, \underline{8}, \underline{9}, 9]\}$, the underscored vertices belong to G_1.

5.2.2 The MSP Algorithm

Algorithm 3 shows the MSP algorithm. It takes the inputs and goes through three steps of *(1)* vertex partitioning and partition pairing with the objective to aggregate iterations without joining the DAGs of the inputs kernels; *(2)* merging and slack vertex assignment to reduce synchronization and to balance workloads; and *(3)* packing to improve locality.

Vertex Partitioning and Partition Pairing.

The first step of MSP partitions one of the input DAGs G_1 or G_2, and then uses that partitioning to partition the other DAG. The created partitions are stored in $\mathcal{V}$. Partitioning the joint DAG is complex and might not be efficient because of the significantly larger number of edges and vertices added compared to the individual DAG of each kernel. Instead, MSP ignores the dependencies across kernels and first creates a partitioning from one of the DAGs with the help of *vertex partitioning*. Then the other DAG is partitioned using a *partition pairing* strategy. The DAG that is

partitioned first is the head DAG and the other is the tail DAG. *A head DAG choice strategy* is used to select the head DAG.

Vertex partitioning. MSP uses the LBC DAG partitioner [28] to construct a partitioning of the head DAG in lines 2 and 11 of Algorithm 3 by calling the function `LBC`. The resulting partitioning has a set of disjoint s-partitions. Each s-partition contains k disjoint w-partitions which are balanced using vertex weights. Disjoint w-partitions ensure all w-partitions within s-partitions are independent. The created partitions are stored in a two-dimensional list H using `list`, e.g. w-partition w_j of s-partition s_i is stored in H_{ij}.

Partition pairing. The algorithm then partitions the tail DAG with *forward pairing*, if G_1 is the head DAG, or with *backward pairing*, if G_2 is the head DAG. With the pairing strategy, some of the partitions of the tail DAG are paired with the head DAG partitions. Pair-partitions are *self-contained* so that they execute in parallel if assigned to the same s-partition. The created partitions are put in the fused partitioning $\mathcal{V}$ to be used in step two. The following first describes the condition for partitions to be self-contained and then explains the forward and backward pairing strategies.

Pair partitions H_{ij} and T_{ij} are called self-contained if all reachable vertices from a breadth first search (BFS) on $\forall v \in H_{ij} \cup T_{ij}$ through vertices of G_1 and G_2 are in $H_{ij} \cup T_{ij}$. Self-contained pair partition (H_{ip}, T_{ip}) and pair partition (H_{iq}, T_{iq}) can execute in parallel without synchronization if in the same wavefront i, i.e. $\forall 1 \leq i \leq b \wedge (1 \leq p, q \leq m_i)$. Partitions that do not satisfy this condition create synchronizations in the final schedule.

The backward pairing strategy visits every partition $H_{i,j}$ and performs a BFS (line 5) from vertex $v_l \in H_{i,j}$ to its dependent vertices in G_1 which are reachable through F_l. Reachable vertices are stored in T_{ij}. The partitions in H and T are assigned a w- and s-partition and are then put in the fused partitioning $\mathcal{V}$ (via `add` in line 6). The assigned s- and w-partitions for H_{ij} are s_{i+1} and w_j respectively, i.e. $\mathcal{V}_{s_{i+1},w_j}$. T_{ij} should be executed before H_{ij} thus is placed in s-partition s_i or $\mathcal{V}_{s_i,w_{m_i+1}}$, where m_i is number of w-partitions in $\mathcal{V}_{s_i}$ at this point. If a vertex in $H_{i,j}$ depends on more than one vertex in G_1, some vertices are replicated in different T partitions. While replication leads to redundant computation, it ensures that the pair partition $(H_{i,j}, T_{i,j})$ is self-contained because vertices that depend on the vertices in $H_{i,j}$ will be included in $T_{i,j}$. MSP performs fusion only if profitable, hence fusion is disabled (by setting `fusion` to `False`) if the number of redundant computations go beyond a threshold. This threshold is $2 \times (|V_1| + |V_2|)$ in line 9 and is defined as the sum of vertices of both DAGs.

The forward pairing strategy iterates over every partition $H_{i,j}$ and performs a BFS from vertex $v_l \in H_{i,j}$ to its reachable vertices in G_2 through F_l^T, see lines 12–18 in Algorithm 3. The list of reachable vertices are stored in $T_{i,j}$ via `BFS` in line 14. If a vertex v_m in $T_{i,j}$ depends on vertex v_l in G_1 and v_l does not exist in $H_{i,j}$ then v_m should be removed to ensure $(H_{i,j}, T_{i,j})$ is self contained. The `remove_uncontained` function in line 15 removes vertex v_m and puts it in partition $U_{i,j}$. Finally, the created partitions are assigned to the fused partitioning $\mathcal{V}$ via `add` in line 16 as follows: $\mathcal{V}_{s_i,w_j} = H_{i,j}$, $V_{s_{i+1},w_{m_{i+1}+1}} = T_{i,j}$, $V_{s_{i+1},w_{m_{i+1}+1}} = U_{i,j}$.

The head DAG choice. MSP chooses the DAG with edges as the head DAG to improve locality. Locality is improved because the head DAG is partitioned with LBC. LBC creates well-balanced partitions with good locality when applied to DAGs with edges. Selecting G_2 as the head DAG reduces inspector overhead. If both G_1 and G_2 are DAGs of kernels with dependency, then G_2 is chosen as the head DAG to reduce inspector overhead. When G_2 is partitioned first, MSP chooses backward pairing which is more efficient compared to forward pairing. Forward pairing traverses F and its transpose F^T and thus performs $2 * nnz_F + 2 * n$ operations where nnz_F is the number of nonzeros in F. However, backward pairing only traverses F and performs $nnz_F + n$ operations.

Example. Figures 5.5b shows the output of MSP after the first step for the inputs in Figure 5.2b. MSP chooses G_1 as the head DAG because it has edges ($|E_1| > 1$), G_2 has no edges. In vertex partitioning, G_1 is partitioned with LBC to create up to three w-partitions (because $r = 3$) per s-partition. The created partitions are shown in Figure 5.5a and are stored in H. The first s-partition $\mathcal{V}_{s_1}$ is stored in H_1 and its three w-partitions are indexed with $H_{1,1}$, $H_{1,2}$, and $H_{1,3}$. Similarly, $\mathcal{V}_{s_2}$ is stored H_2 and its only w-partition is in $H_{2,1}$. Figure 5.5b shows the output of partition pairing. Since G_1 is the head DAG, MSP uses forward pairing and performs a BFS from each partition in H to create self-contained pair partitions stored in T. For example, a BFS from $H_{1,1} = \{\underline{1}, \underline{2}, \underline{3}, \underline{4}\}$ creates $T_{1,1} = \{1, 2, 3, 4\}$. Since $T_{1,1}$ and $H_{1,1}$ are self-contained, no vertices are removed from $T_{1,1}$ and thus $U_{1,1} = \emptyset$. Finally, MSP puts $H_{1,1}$ and $T_{1,1}$ in $\mathcal{V}_{s_1,w_1}$ and $\mathcal{V}_{s_2,w_2}$ respectively, and adds $(\mathcal{V}_{s_1,w_1}, \mathcal{V}_{s_2,w_2})$ to $\mathcal{V}.pairs$. The final partitions and pairings as shown in Figure 5.5b are: $\mathcal{V} = [\{H_{1,1}, H_{1,2}, H_{1,3}\}, \{H_{2,1}, T_{1,1}, T_{1,2}, T_{1,3}\}, \{T_{2,1}\}] = [\{\{\underline{1}, \underline{2}, \underline{3}, \underline{4}\}, \{\underline{5}, \underline{6}\}, \{\underline{7}, \underline{8}, \underline{9}\}\}, \{\{\underline{10}, \underline{11}\}, \{1, 2, 3, 4\}, \{5, 6\}, \{7, 8, 9\}\}, \{\{10, 11\}\}]$ and the pairing information is: $\mathcal{V}.pairs = \{(\mathcal{V}_{s_1,w_1}, \mathcal{V}_{s_2,w_2}), (\mathcal{V}_{s_1,w_2}, \mathcal{V}_{s_2,w_3}), (\mathcal{V}_{s_1,w_3}, \mathcal{V}_{s_2,w_4}), (\mathcal{V}_{s_2,w_1}, \mathcal{V}_{s_3,w_1})\}$.

Merging and Slack Vertex Assignment.

The second step of MSP reduces the number of synchronizations by merging some of the pair partitions in a *merging* phase. It also improves load balance by dispersing vertices across partitions using *slacked vertex assignment*.

Slack definitions: A vertex v can always run in its wavefront number $l(v)$. However, the execution of vertex v can sometimes be postponed up to $SN(v)$ wavefronts without having to move its dependent vertices to later wavefronts. $SN(v)$ is the slack number of v and is defined as $SN(v) = P_G - l(v) - height(v)$ where $height(v)$ is the maximum path from a vertex v to a sink vertex (a sink vertex is a vertex without any outgoing edge), P_G is the critical path of G, and $l(v)$ is the wavefront number of v. A vertex with a positive slack number is a *slack vertex*. To compute vertex slack numbers efficiently, instead of visiting all vertices, MSP iterates over partitions and computes the slack number of each partition in the partitioned DAG, i.e. *partition slack number*. The computed slack number for a partition is assigned to all vertices of the partition. As shown in line 20 of Algorithm 3, all partition slack numbers of $\mathcal{V}$ are computed via `slack_info` and are stored in $\mathcal{S}$. For example, because vertices in $\mathcal{V}_{s_2,w_3}$ can be postponed one wavefront, from s-partition 2 to 3, their slack number is 1. Vertices in w-partitions $\mathcal{V}_{s_2,w_1}$ and $\mathcal{V}_{s_3,w_1}$ can not be moved because their slack numbers are zero.

Merging. MSP finds pair partitions with partition slack number of zero and then merges them as shown in lines 21-23. Since pair partitions are self contained, merging them does not affect the correctness of the schedule. Algorithm 3 visits all pair partitions (w, w') in $\mathcal{V}.pairs$ and merges them using the `merge` function in line 22 if their slack numbers are zero, i.e. $SN(w) = 0$ and $SN(w') = 0$. The resulting merged partition is stored in $\mathcal{V}$ in place of the w-partition with the smaller s-partition number.

Slacked vertex assignment. The algorithm then uses slacked vertex assignment to approximately load balance the w-partitions of an s-partition using a cost model. The cost of w-partition $w \in \mathcal{V}_{s_i}$ is defined as $cost(w) = \sum_{v \in w} c(v)$. A w-partition is balanced if the maximal difference of its cost and the cost of other w-partitions in its s-partition is smaller than a threshold ϵ. The maximal difference for a w-partition inside a s-partition is computed by subtracting its cost from the cost of the w-partition (from the same s-partition) with the maximum cost.

MSP first removes all slacked vertices $\mathcal{S}$ from the fused partitioning $\mathcal{V}$ in line 24. It then goes over every s-partition i and w-partition j and balances $\mathcal{V}_{s_i,w_j}$ by assigning a slacked vertex to it where possible. W-partition $\mathcal{V}_{s_i,w_j}$ becomes balanced with vertices from its pair partition using the function `balance_with_pair` in line 27. If $\mathcal{V}_{s_i,w_j}$ is still imbalanced, `balance_with_slacks` in line 28 balances the w-partition using the

slacked vertices $v_l \in \mathcal{S}$ that satisfy the following condition $l(v_l) < i < (l(v_l)+SN(v_l))$. Slack vertices in $\mathcal{S}$ that depend on each other are dispersed as a group to the same w-partition for correctness. In line 30, slacked vertices in $\mathcal{S}$ that are not postponed to later s-partitions are evenly divided between the w-partitions of the current s-partition ($\mathcal{V}_{s_i}$) using the `assign_even` function.

Example. Figure 5.5d shows the output of the second step of MSP from the partitioning in Figure 5.5b. First pair partitions ($\mathcal{V}_{s_2,w_1}, \mathcal{V}_{s_3,w_1}$), shown with red dash-dotted circles in Figure 5.5b, are merged because their slack numbers are zero. The resulting merged partition is placed in $\mathcal{V}_{s_2,w_1}$ to reduce synchronization as shown in Figure 5.5c. Then slacked vertex assignment balances the w-partitions in Figure 5.5c. The balanced partitions are shown in Figure 5.5d. The slacked vertices S, are shown with dotted blue circles in Figure 5.5c. The w-partitions in $\mathcal{V}_{s_1}$ are balanced using vertices of their pair partitions, e.g. the yellow dash-dotted vertices 5 and 6 are moved to w_2 in $\mathcal{V}_{s_1}$ as shown in Figure 5.5d. `balance_with_slacks` is used to balance partitions in $\mathcal{V}_{s_2}$. This is because the vertices in S do not belong to the pair partitions of the w-partitions in $\mathcal{V}_{s_2}$. However, since the slack vertices in S can execute in either s-partition two or three because they are from s-partition one and have a slack number of one, they are used to balance the w-partitions in $\mathcal{V}_{s_2}$.

Packing.

The third step of MSP reorders the vertices inside a w-partition to improve data locality for a thread within each kernel or between the two kernels. The previous steps of the algorithm create w-partitions that are composed of vertices of one or both kernels however the order of execution is not defined. Using the reuse ratio, the order at which the nodes in a w-partition should be executed is determined with a packing strategy. MSP has two packing strategies: *(i)* in interleaved packing, the vertices of the two DAGs in a w-partition are interleaved for execution and *(ii)* in separated packing the vertices of each kernel are executed separately. Interleaved packing improves temporal locality between kernels while separated packing enhances spatial and temporal locality within kernels. When the reuse ratio is greater than one, in line 32 of Algorithm 1 function `interleaved_pack` is called to interleave iterations of the two kernels based on F. Otherwise, `separated_pack` is called (line 33) to pack iterations of each kernel separately.

Example. Figure 5.2e shows the output of MSP's third step from the partitioning in Figure 5.5d. Since the reuse ratio is smaller than one separated packing is chosen thus $\mathcal{V}_{s_2,w_1}$ is stored as $\mathcal{V}_{s_2,w_1} = \{[\underline{10}, \underline{11}, 10, 11]\}$. Vertices are ordered to keep dependent iterations of SpTRSV and consecutive iterations SpMV next to each other.

Table 5.1: The list of sparse matrices.

ID	Name	Nonzeros	ID	Name	Nonzeros
1	Flan_1565	117.4×10^6	**5**	Emilia_923	41×10^6
2	bone010	71.7×10^6	**6**	StocF-1465	21×10^6
3	Hook_149	60.9×10^6	**7**	af_0_k101	17.6×10^6
4	af_shell10	52.3×10^6	**8**	ted_B_unscal	0.14×10^6

To use MSP on more than two loops, in lexicographical order, the first two loops are fused with MSP. Then the remaining loops are added to the fused schedule, one at a time. The DAG being added to the fused schedule will be the tail DAG. The extension requires a strategy to select the most profitable loops to be fused which is not explored here.

5.3 Experimental Results

We compare the performance of sparse fusion to MKL [192] and ParSy [28], two state-of-the-art tools that accelerate individual sparse kernels, which we call unfused implementations. Sparse fusion is also compared to three fused implementations that we create. To our knowledge, sparse fusion is the first work that provides a fused implementation of sparse kernels where at least one kernel has loop-carried dependencies. For comparison, we also create three fused implementations of sparse kernels by applying LBC, DAGP, and a wavefront technique to the joint DAG of the two input sparse kernels and create a schedule for execution using the created partitioning, the methods will be referred to as fused LBC, fused DAGP, and fused wavefront in order.

Table 5.2: The list of kernel combinations. CD: loops with carried dependencies, SpIC0: Sparse Incomplete Cholesky with zero fill-in, SpILU0: Sparse Incomplete LU with zero fill-in, DSCAL: scaling rows and columns of a sparse matrix.

ID	Kernel combination	Operations	Dependency DAGs	Reuse Ratio
1	SpTRSV CSR - SpTRSV CSR	$x = L^{-1}y, z = L^{-1}x$	CD - CD	$\frac{2n+2size_L}{max(2n+size_L,size_L+2n)} \geq 1$
2	SpMV CSR - SpTRSV CSR	$y = Ax$, $z = L^{-1}y$	Parallel - CD	$\frac{2n}{max(2n+size_L,size_A+2n)} < 1$
3	DSCAL CSR - SpILU0 CSR	$LU \approx DAD^T$	Parallel - CD	$\frac{2size_A}{max(size_A,size_A+2n)} \geq 1$
4	SpTRSV CSR - SpMV CSC	$y = L^{-1}x$, $z = Ay$	CD - Parallel	$\frac{2n}{max(2n+size_L,size_A+2n)} < 1$
5	SpIC0 CSC - SpTRSV CSC	$LL^T \approx A$, $y = L^{-1}x$	CD - CD	$\frac{2size_L}{max(size_L,size_L+2n)} \geq 1$
6	SpILU0 CSR - SpTRSV CSR	$LU \approx A$, $y = L^{-1}x$	CD - CD	$\frac{2size_A}{max(size_A,size_L+2n)} \geq 1$
7	DSCAL CSC - SpIC0 CSC	$LL^T \approx DAD^T$	Parallel - CD	$\frac{2size_L}{max(size_L,size_L+2n)} \geq 1$

Setup. The set of symmetric positive definite matrices listed in Table 5.1 are used for experimental results. The matrices are from [41] and with real values in double precision. The test-bed architecture is a multicore processor with 12 cores described as Haswell-EP in Table 2.1[3]. All generated codes, implementations of different approaches, and library drivers are compiled with GCC v.7.2.0 compiler and with the `-O3` flag. Matrices are first reordered with METIS [103] to improve parallelism.

We compare sparse fusion with two unfused implementations where each kernel is optimized separately: *I. ParSy* applies LBC to DAGs that have edges. For parallel loops, the method runs all iterations in parallel. LBC is developed for L-factors [37] or chordal DAGs. Thus, we make DAGs chordal before using LBC. *II. MKL* uses Intel MKL [192] routines with MKL 2019.3.199 and calls them separately for each kernel.

Sparse fusion is also compared to three fused approaches all of which take as input the *joint DAG*; the joint DAG is created from combining the DAGs of the input kernels using the inter-DAG dependency matrix F. We then implement three approaches to build the fused schedule from the joint DAG: *I. Fused wavefront* traverses the joint DAG in topological order and builds a list of wavefronts that represent vertices of both DAGs that can run in parallel. *II. Fused LBC* applies the LBC algorithm to the joint DAG and creates a set of s-partitions each composed of independent w-partitions. Then the s-partitions are executed sequentially and w-partitions inside an s-partition are executed in parallel. LBC is taken from ParSy and its parameters are tuned for best performance. The joint DAG is first made chordal and then passed to LBC. *III. Fused DAGP* applies the DAGP partitioning algorithm to the joint DAG and then executes all independent partitions that are in the same wavefront in parallel. DAGP is used with METIS for its initial partitioning, with one run (`runs=1`) and the remaining parameters are set to default.

The list of sparse kernel combinations investigated are in Table 5.2. To demonstrate sparse fusion's capabilities, the sparse kernels are selected with different combinations of storage formats, i.e. CSR and compressed sparse column (CSC) storage, different combinations of parallel loops and loops with carried dependencies, and a variety of memory access pattern behaviour. For example, combinations of SpTRSV, $Lx = b$ and SpMV are main bottlenecks in conjugate gradient methods [203, 18], GMRES [30], Gauss-Seidel [152]. Preconditioned Krylov methods [78] and Newton solvers [168] frequently use kernel combinations 3, 5, 6, 7. The s-step Krylov solvers [24] and s-step optimization methods used in machine learning [168] provide even more opportunities to interleave iterations. Thus, they use these kernel combinations

[3]Results for the Skylake processor in Table 2.1 with 24 cores are shown in Section A.2.

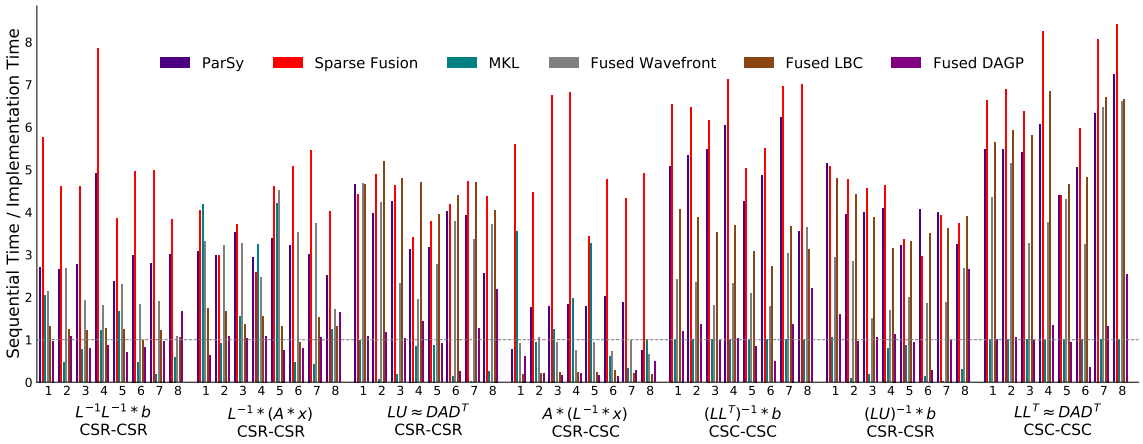


Figure 5.6: Performance of different implementations shown with speedup from dividing baseline time by implementation time.

significantly more than their classic formulations.

Table 5.3: The achieved GFLOP/s for the baseline code for the kernel combinations in Table 5.2 and for matrices in Table 5.1.

	Kernel Combination ID						
Matrix ID	1	2	3	4	5	6	7
1	1.52	1.54	0.45	1.55	0.61	0.43	0.61
2	1.5	1.54	0.45	1.54	0.61	0.45	0.61
3	1.4	1.45	0.47	1.45	0.48	0.50	0.47
4	1.47	1.48	0.72	1.49	0.50	0.77	0.47
5	1.42	1.47	0.45	1.47	0.51	0.46	0.49
6	0.91	1.14	0.17	1.14	0.33	0.18	0.32
7	1.47	1.50	0.73	1.49	0.49	0.77	0.48
8	1.41	1.70	0.89	1.70	0.44	0.76	0.42

Sparse Fusion's Performance. Figure 5.6 shows the performance of the fused code from sparse fusion, the unfused implementation from ParSy and MKL, and the fused wavefront, fused LBC, and fused DAGP implementations. All execution times are normalized over a *baseline*. The baseline is obtained by running each kernel individually with a sequential implementation. The floating point operations per second (FLOP/s) for each implementation can be obtained by multiplying the baseline FLOP/s from Table 5.3 with the speedups in Figure 5.6. The sparse fusion's fused code is on average 1.6× faster than ParSy's executor code and 5.1× faster than MKL across all kernel combinations. Even though sparse fusion is on average 11.5× faster than MKL for ILU0-TRSV, since ILU0 only has a sequential implementation in MKL, the speedup of this kernel combination is excluded from the average speedups. The fused code from sparse fusion is on average 2.5×, 5.1×, and 7.2× faster than in order fused wavefront, fused LBC, and fused DAGP. Obtained speedups of sparse fusion over ParSy (the fastest unfused implementation) for SpILU0-SpTRSV and SpIC0-

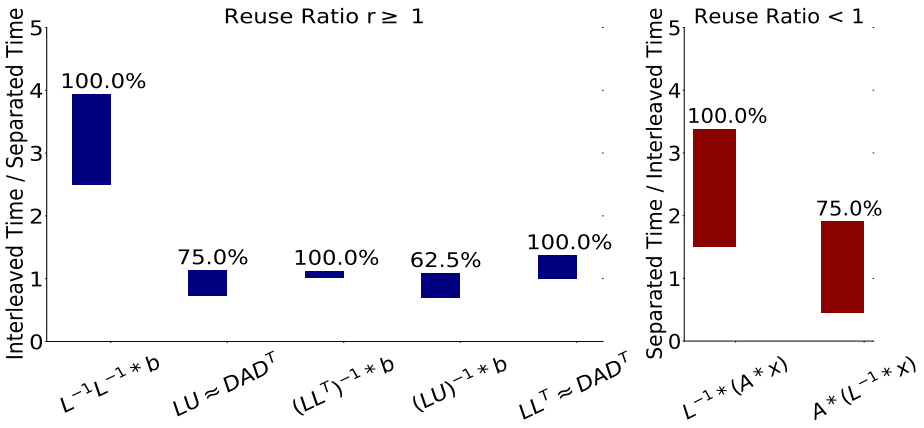


Figure 5.7: The range of speedup for all matrices achieved as a result of using interleaved packing vs. separated packing. The labels on bars show how often the choice of packing strategy made by sparse fusion leads to performance improvement.

SpTRSV is lower than other kernel combinations. Because SpIC0 and SpILU0 have a high execution time, when combined with others sparse kernels with a noticeably lower execution time, the realized speedup from fusion will not be significant.

Locality in Sparse Fusion. Figure 5.7 shows the efficiency of the two packing strategies to improve locality. The effect of the packing strategy is shown for kernel combinations with a reuse ratio smaller and larger than one as shown in Table 5.2. Kernel combinations 1, 3, 5, 6, and 7 share the sparse matrix L and thus have a reuse ratio larger than one while combination 2 and 4 only share vector y leading to a reuse ratio lower than one. Figure 5.7 shows the range of speedup over all matrices for the selected packing strategy versus the other other packing method for each combination. As shown, the selected packing strategy in sparse fusion improves the performance in 88% of kernel combinations and matrices and provides 1-3.9× improvement in both categories.

Figure 5.8 shows the average memory access latency [85] of sparse fusion, the fastest unfused implementation (ParSy), and the fastest fused partitioning-based implementation (Fused LBC) for all kernel combinations normalized over the ParSy average memory access latency (shown for matrix *bone010* as example, other matrices exhibit similar behavior). The average memory access latency is used as a proxy for locality and is computed using the number of accesses to L1, LLC, and TLB measured with PAPI performance counters [180].

For kernels 1, 3, 5, 6, and 7 where the reuse ratio is larger than one, the memory access latency of ParSy is on average 1.3× larger than that of sparse fusion. Because of their high reuse ratio, these kernels benefit from optimizing locality between kernels made possible via interleaved packing. ParSy optimizes locality in each kernel

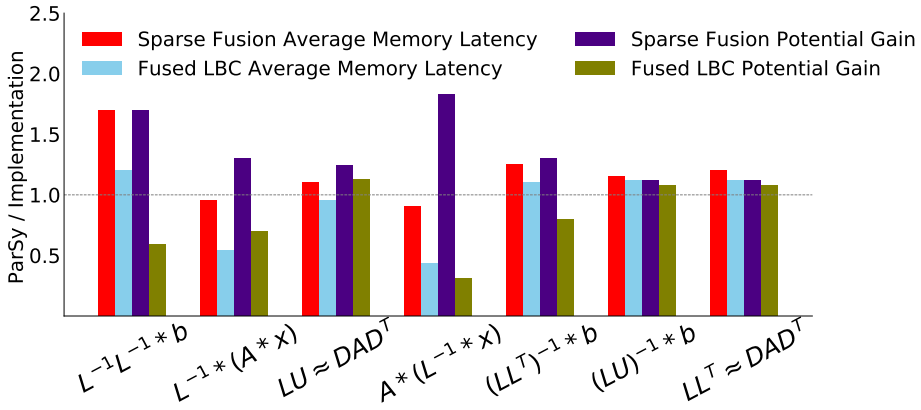


Figure 5.8: Average memory access time and the OpenMP potential gain for matrix *bone010*. The legends show the implementation, values are normalized over ParSy.

individually. When applied to the joint DAG, LBC can potentially improve the temporal locality between kernels and thus there is only a small gap between the memory access latency of sparse fusion and that of fused LBC. For kernels 2 and 4 where the reuse ratio is smaller than one, the gap between the memory access latency of sparse fusion and fused LBC is larger than the gap between the memory access latency of sparse fusion and ParSy. Sparse fusion and ParSy both improve data locality within each kernel for these kernel combinations.

Load Balance and Synchronization in Sparse Fusion. Figure 5.8 shows the OpenMP potential gain [165] of sparse fusion, ParSy, and Fused LBC for all kernel combinations normalized over ParSy's potential gain (shown for matrix *bone010* as example, but all other matrices in Table 5.1 follow similar behavior.) The OpenMP potential gain is a metric in Vtune [205] that shows the total parallelism overhead, e.g. wait-time due to load imbalance and synchronization overhead, divided by the number of threads. This metric is used to measure the load imbalance and synchronization overhead in ParSy, fused LBC, and sparse fusion.

Kernel combinations 2 and 4 have slack vertices that provide opportunities to balance workloads. For example, for matrices shown in Table 5.1, between 35-76% vertices can be slacked thus the potential gain balance of ParSy is 1.6× larger than sparse fusion and 2.4× lower than fused LBC. ParSy can only improve load balance using the workloads of an individual kernel. As shown in Figure 5.1, for the kernel combination 5, the joint DAG has a small number of parallel iterations in final wavefronts that makes the final s-partitions of the LBC fused implementation imbalanced (a similar trend exists for kernel combination 6). For these kernel combinations, the code from sparse fusion has on average 33% fewer synchronization barriers compared to ParSy due to merging. For kernel combinations 1, 2, 3, 4, and 7 the potential gain

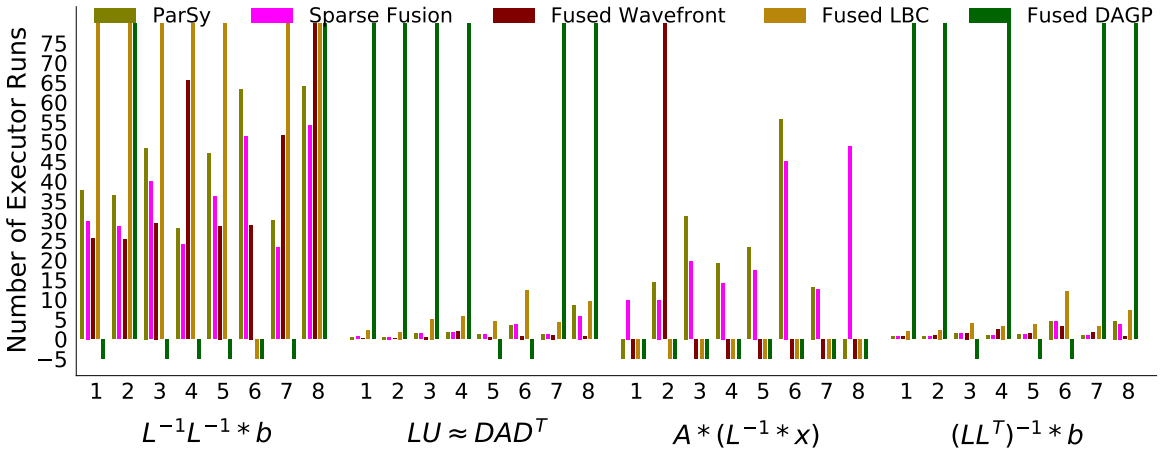


Figure 5.9: The number of executor runs to amortize inspector cost. Values are clipped between -5 and 80. (lower is better)

in sparse fusion is 1.3× less than that of ParSy. Merging in sparse fusion reduces the number of synchronizations in the fused code on average 50% compared to that of ParSy.

Inspector Time. Figure 5.9 shows the number of times that the executor should run to amortize the cost of inspection for implementations that have an inspector. For space only combinations 1, 3, 4, and 5 are shown, others follow the same trend. The number of executor runs (NER) that amortize the cost of inspector for an implementation is calculated using $\frac{Inspector\ Time}{Baseline\,Time - \ Executor\ Time}$. The *baseline* time is obtained by running each kernel individually with a sequential implementation, the inspector and executor times belong to the specific implementation. The fused LBC implementation has a NER of 3.1-745. The high inspection time is because of the high cost of converting the joint DAG into a chordal DAG, typically consuming 64% of its inspection time. The NER of the fused DAGP implementation is either negative or higher than 80. The fused wavefront implementation sometimes has a negative NER because the executor time is slower than the baseline time. As shown, sparse fusion and fused wavefront have the lowest NER amongst all implementations. Sparse fusion's low inspection time is due to pairing strategies that enable partitioning one DAG at a time. Kernel combinations such as, SpIC0-TRSV and SpILU0-TRSV only need one iteration to amortize the inspection time and SpTRSV-SpMV, SpTRSV-SptRSV, and SpMV-SpTRSV need between 11-50 iterations. Sparse kernel combinations are routinely used in iterative solvers in scientific applications. Even with preconditioning, these solvers typically converge to an accurate solution after ten of thousands of iterations [18, 106, 136], hence amortizing the overhead of inspection.

5.4 Related work

A number of libraries and inspector-executor frameworks provide parallel implementations of fused sparse kernels with no loop-carried dependencies such as, two or more SpMV kernels [91, 123, 128, 4, 151] or SpMV and dot products [203, 43, 4, 65, 3, 151]. The formulation of s-step Krylov solvers [24] has enabled iterations of iterative solvers to be interleaved and hence multiple SpMV kernels are optimized simultaneously via replicating computations to minimize communication costs [91, 123, 128, 168]. Sparse tiling [171, 108, 176, 172, 173] is an inspector executor approach that uses manually written inspectors [171, 173] to group iteration of different loops of a specific kernel such as Gauss-Seidel [173] and Moldyn [171] and is generalized for parallel loops without loop-carried dependencies [176, 108]. Sparse fusion optimizes combinations of sparse kernels where at least one of the kernels has loop-carried dependencies.

Chapter 6

Adaptive Sparsity Pattern in Quadratic Programming

Sympiler generates optimized and parallel code for a kernel or a combination of two sparse kernels as discussed in Chapters 3-5. However, the sparsity structures in physical simulations and numerical optimization alter with typically small and infrequent changes and often affect a few rows or columns of the matrix. Also, typically the domain experts and operators have a priori knowledge of the pattern changes. For example, in minimizing a quadratic objective with linear constraints, i.e., *quadratic programming*, in each iteration the sparsity pattern of the linear system changes, with the changes being determined by the constraint set which is known prior to solving. Solving the linear system from scratch would make the algorithm take a very long time and often not scalable. In this chapter, an indefinite solver and an update/downdate technique, SoMod are presented that enable Sympiler to reuse information from previous solves to improve the runtime. The application of SoMod in an active-set QP solver, called NASOQ is tested and compared with other QP solvers. The content of this chapter is published in [30].

6.1 Introduction

Solving a quadratic program (QP) is a core numerical task critical in domains spanning geometry processing [202, 177, 53], animation [95, 149], physical simulation [17, 56, 15, 178], robotics [135], machine learning [9, 1], engineering, and design [58]. Unfortunately, available QP solvers are often neither accurate nor robust enough for many applications [104, 201, 164, 199, 202], necessitating heuristics, approximations and/or multiple failsafe backups to succeed.

A long-standing challenge then has been to provide a single, unified QP solver

that is 1) accurate, 2) efficient, and 3) scalable. By accurate we mean that the QP solver converges to all reasonable requested accuracies; by efficient we mean that it converges rapidly in wall-clock time; and by scalable we mean that it efficiently converges across both large- and small-sized QP instances. As we show in Section 6.5, available QP solver libraries generally succeed for some subsets of QPs, while often failing or becoming impractically slow to achieve success for others. To make matters worse, in many cases, given the algorithms employed, it is not possible to predict in advance when a QP method will succeed or fail per QP problem instance [201].

The key challenge for solving a QP is in identifying an *active set* [59]. An active set is a subset of a QP's linear inequality constraints that are treated as equalities at optimality. All other inequalities can then effectively be safely ignored. If an active set is found, a QP problem instance then reduces to solving a much easier QP subject to just its active constraints set as equalities.

Algorithms for solving large-scale QPs generally treat the entire constraint set as approximately "active" with barrier terms penalizing all constraint violations simultaneously. This allows the application of large-scale, general-purpose sparse linear solvers, but generally comes at the cost of uncertainty in the active set and degraded solution accuracy. On the other hand, to address accuracy, many other QP algorithms employ *active-set methods*. These are a range of methods that iteratively explore and test active-set proposals. Details vary across methods but in all cases each iteration requires solving large numbers of reduced QPs. Each reduced QP is solved subject to a different set of proposed active constraints treated as equalities. In turn, solving these many reduced QPs accurately and efficiently is the computational crux of active-set methods. This amounts to solving at each instance an indefinite linear system for equality constrained optimality conditions – a Karush-Kuhn-Tucker (KKT) system [23, 59]. Current solutions employed rely either on accurate linear solvers that work well for small systems but are too expensive for repeated solves of new large, sparse problems, or else rely on less expensive but also less accurate methods for solving linear systems that once again unacceptably reduce accuracy [170].

To address these issues we construct the Numerically Accurate Sparsity-Oriented QP Solver (NASOQ), a new, general-purpose, active-set algorithm for the accurate, efficient, and scalable solution of QPs. NASOQ is built upon three core contributions:

- LBL: a new LDL factorization algorithm for the fast, accurate factorization and update of sparse symmetric indefinite systems including those that arise in KKT problems;
- SoMod: a new sparsity-oriented row modification method that enables fast fac-

torization for KKT matrix changes via efficient updates of previously computed factors; and

- Two new QP solvers that extend the Goldfarb-Idnani (GI) active-set strategy [70] by application of LBL and SoMod to enable user-exposed trade-offs between speed and accuracy for large and sparse QP problems.

SoMod is a new algorithm designed to enable the rapid and accurate solutions of the many successively-updated KKT systems encountered during active-set QP solves. At start of QP solves, SoMod performs an initial symbolic analysis of a KKT system containing all constraints, then utilizes this information for both the initial factorization (which includes only the equality constraints) as well as subsequent factorizations (which include proposed active sets). By precomputing symbolic information, SoMod enables efficiently updating the factorization when constraints are added or removed from the proposed active set.

To compute each initial indefinite factorization for SoMod, we construct LBL, a novel implementation of the LDL factorization using Load-Balanced Level Coarsening [28] for parallelization. LBL provides state-of-the-art performance for solving indefinite KKT systems while enabling precomputation of the required symbolic analysis for subsequent factorization updates.

With these building blocks in place we construct and analyze NASOQ via a pair of new active-set QP algorithms. To consistently evaluate NASOQ with both prior and future QP methods we also introduce a new benchmark set composed of both practical, real-world stress-test QPs taken from a wide range of geometry, simulation, and design applications as well as prior QP benchmarks [121, 95, 194, 160, 109, 96]. As we demonstrate in Section 6.5, across a range of requested accuracies in this benchmark NASOQ obtains consistent accuracy by converging for 99.5% of benchmark problems while the best convergence across competing state-of-the-art solvers is 94%. At the same time NASOQ remains most efficient by providing an average speedup of 1.7×–24.8× across requested accuracy ranges compared to the fastest competing times across compared QP solvers. Please see Section 6.5 for details of our analysis.

6.2 Problem Statement and Preliminaries

We focus on the solution of convex quadratic programming problems to find the linearly constrained minimizers of quadratic energies. In full generality our problem

then is

$$\min_x \frac{1}{2} x^T H x + q^T x \text{ s.t. } Ax = b,\ Cx \leq d \tag{6.1}$$

where the unknown minimizer $x \in \mathbb{R}^n$ is constrained by linear equality constraints $Ax = b$ and inequality constraints $Cx \leq d$. Note that in many cases we may have only inequality or equality constraints. However, in the following, without loss of generality, we consider the full mixed case. Here the symmetric matrix H is, either by construction or standard user regularization [154, 71], a positive-definite matrix. The QP in (6.1) is then strictly convex. In applications, the matrices H, A, and C are often large and sparse. By sparse we mean that the majority of matrix entries are zero, e.g., we have an average of 98% zero entries in our benchmark examples.

Unlike the solution of symmetric linear systems (or equivalently, unconstrained quadratic energies) the optimality conditions, and thus the accuracy of a QP solution, are much more complex to evaluate. Optimality of (6.1) is given by the specialized Karush-Kuhn-Tucker (KKT) conditions[1] [196, 59]

$$\begin{aligned} Hx + q + A^T y + C^T z &= 0 \\ Ax - b &= 0 \\ 0 \leq z \perp\ d - Cx &\geq 0. \end{aligned} \tag{6.2}$$

where y and z are the QP problem's Lagrange multipliers [196].

6.2.1 Accuracy

Applications require controllable quality and thus controllable accuracy for solutions to the QP problem. The accuracy of a QP solution is evaluated by reduction of four corresponding measures[2]

$$\text{Primal-feasibility: } \left\| \Big((Ax-b)^T, (\max(\mathbf{0}, Cx-d))^T \Big)^T \right\| < \epsilon_f, \tag{6.3}$$

$$\text{Stationarity: } \|Hx + q + A^T y + C^T z\| < \epsilon_s, \tag{6.4}$$

$$\text{Complementarity: } \|z \odot (Cx - d)\| < \epsilon_c, \tag{6.5}$$

$$\text{Non-negativity: } \|\min(\mathbf{0}, z)\| < \epsilon_n. \tag{6.6}$$

In the following we design NASOQ and analyze QP methods on their ability to drive all four of these measures (∞-norm) below a common, maximum error threshold accuracy: $\epsilon \geq \max(\epsilon_f, \epsilon_s, \epsilon_c, \epsilon_n)$. While necessary accuracies for each of the four

[1] Here $x \perp y$ is the *complementarity condition* $x_i y_i = 0,\ \forall$ corresponding entries i in vectors x and y.
[2] Here $\odot$ is the *Hadamard* (element-wise) product.

measures certainly change per application, a desirable goal for a general-purpose QP algorithm is to solve every reasonable problem to any requested accuracy. Here we design for general-purpose QP problems and so do not predict a priori what measures are most important. Thus we evaluate fitness by asking each solve to drive all measures below ϵ.

Primal-feasibility measures constraint satisfaction. Applying the ∞-norm gives the worst violation of the enforced constraints by a given solution. In many applications constraints are invariants that need to be satisfied such as positive volume, non-penetration, or structural feasibility. Errors in constraint satisfaction lead to unacceptable failures and constraint drift in applications that depend on constraint resolution at each call of a QP solve.

Stationarity measures the balance between energy and constraint gradients. This is critical for stable and accurate solutions. For example, in structural engineering applications stationarity measures how well force balance is modeled, while in dynamic simulations stationarity measures how well the equations of motion are satisfied. In many applications even small residuals of stationarity with respect to measured dimensions of the systems can lead to simulation instabilities and blow-ups and/or unacceptable modeling errors for engineering applications.

Complementarity measures the pairwise products of dual variables and their corresponding inequality constraints and is critical for correctly capturing active sets. For example, in multi-body simulations [56, 90] dual variables often represent contact forces while constraints model intersections. Complementarity then encodes the property that contact forces cannot be applied unless objects are touching. Large violations of complementarity can create instabilities and visual artifacts of floating bodies with contact forces artificially applying action at a distance.

Non-negativity then ensures that dual variables are positive. Negative dual variables likewise have serious consequences for stability and quality in applications. Consider, for example, bounded biharmonic weights for deformation skinning are computed via QP solves [95] with resultant weights requiring non-negativity; while similarly for contact problems negative dual variables indicate an unacceptable violation of the "no-velcro" condition – that contact forces should push but not pull [164].

6.2.2 Active-Set KKT System Solutions

We focus on enabling scalable, efficient, and accurate solutions for QPs at all scales. For a given input QP we seek an as-efficient-as-possible solver that will obtain a user-requested accuracy. While state-of-the-art barrier and first-order QP methods

Algorithm 4: Dual-feasible active-set QP solver.

Data: H, q, A, b, C, d
Result: x, y, z
/* Feasibility phase */
1 Initialize $z_0 = 0$; $k = 0$; active-set=$\emptyset$;
2 Solve Equation 6.7 to initialize x_0, y_0;
/* Optimality phase */
3 **while** x_k *is not primal-feasible* **do**
4 Solve Equation 6.8 to compute descent Δx,Δy,Δz;
5 Compute the step length t;
6 **if** $t = \infty$ **then**
7 Problem is unbounded.;
8 **else**
9 Update $x_{k+1}, y_{k+1}, z_{k+1}$ with $\Delta x, \Delta y, \Delta z$;
10 Update the active set;
11 Update the KKT system with the updated active set;
12 **end**
13 $k = k + 1$;
14 **end**
15 x_k, y_k, z_k are optimal;

are promising for scaling to large QP problems, their solutions suffer from degraded accuracy [170] and no general method exists for determining a priori when they will succeed or fail in reaching the requested accuracy [22]. On the other end of the spectrum, active-set QP methods provide high-accuracy QP solutions. However, in order for active-set QP algorithms to reach a targeted accuracy they must also accurately solve a large number of successive indefinite linear systems visited by the algorithm at each inner iteration, which can be computationally expensive.

Active-set methods start with a feasible solution and keep a running set of proposed *active* inequality constraints $\mathcal{W}$ to reach the optimal solution while maintaining feasibility conditions. Active-set methods are then either primal-feasible, preserving the primal-feasibility condition or else are dual-feasible, preserving the non-negativity condition. Here we focus on the Goldfarb-Idnani (GI) [70] strategy. GI is a dual-feasible active-set approach and so enables direct and inexpensive initialization [196].

The high-level pseudocode for the GI algorithm is shown in Algorithm 4. The GI algorithm begins (lines 1–2) by initializing an empty active-set proposal, $\mathcal{W} = \emptyset$ with zero dual variables, $z_0 = 0$. The resulting initial KKT system to solve is then the indefinite linear system,

$$\begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} -q \\ b \end{bmatrix} \tag{6.7}$$

which corresponds to solving a *feasible* QP with just equality constraints applied.

Then, each successive iteration of the GI method (lines 3–14, Algorithm 1) im-

proves the last iterate's solution by updating the active-set proposal $\mathcal{W}$ and so the corresponding active-set constraint matrix $C_{\mathcal{W}}$ and the right-hand side constraint vector c_w. The GI method updates the active set by only adding one or removing one constraint in each successive iteration. Here w is the activated constraint.

The next descent direction for the QP is then determined by solving the *updated* KKT system

$$\begin{bmatrix} H & A^T & C_{\mathcal{W}}^T \\ A & 0 & 0 \\ C_{\mathcal{W}} & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} c_w \\ 0 \\ 0 \end{bmatrix} \tag{6.8}$$

The dual and primal variables of the next iteration are then updated by finding step lengths along the computed descent directions. The step lengths ensure that the activated constraint becomes primal-feasible and all dual variables remain dual-feasible. Thus, in each iteration, both the dual and primal variables corresponding to the constraints in the active set are both non-negative and primal-feasible. Each iteration's linear solve of the updated indefinite KKT system in (6.8) becomes increasingly expensive as QP system sizes and constraint numbers grow. However, at the same time, the GI algorithm requires accurate solutions for each of these successive KKT systems for algorithmic stability and in order to consistently obtain accurate solutions for the overall QP problem [143].

A key observation then is that each update to $\mathcal{W}$ and correspondingly to the matrix in (6.8) is small and specifically requires the update of just a single row in $C_{\mathcal{W}}$. Currently, active-set algorithms leverage this observation with indirect methods [73] that solve the KKT system by adaptively updating it with respect to $\mathcal{W}$ via the application of the QR decomposition and the Schur complement. While indirect methods provide accurate and efficient KKT solutions at small scales, they are unable to take advantage of sparsity. These methods suffer from extensive fill-in[3] due to the QR factorization and the Schur complement form and so do not scale due to slow compute times and memory overhead for QP problems with large numbers of variables and/or large numbers of constraints.

Alternately direct active-set QP methods form the KKT system explicitly and solve it via direct or iterative linear solvers. Direct methods solve each iteration's KKT system via indefinite factorization methods [120] followed by a solve stage. This leads to accurate and scalable solutions but is inefficient due to the repeated cost of recomputing factorizations. Application of iterative methods, e.g., Krylov subspace methods [74], in place of direct solves are not typically performed as it remains challenging to find effective, general-purpose preconditioners for KKT matrices with

[3] *Fill-ins* are additional nonzeros created in the factor during a matrix factorization.

generic active sets [120]. Re-purposing prior factorizations to compute the solution of a modified KKT system is thus a highly attractive direction for combined efficiency and accuracy. However, to our knowledge, no previous solution for indefinite matrix factorization updates exists. The closest possible option we find is CHOLMOD row modification [40], which is an efficient and effective solution designed for symmetric positive definite (SPD) matrices but does not provide accurate and stable solution updates for indefinite KKT systems.

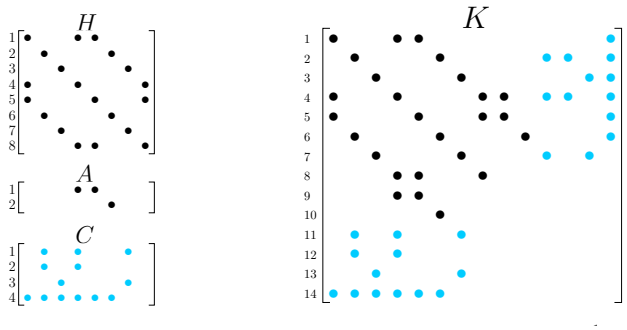
In this dissertation, we focus on enabling accurate, scalable, and efficient QP solutions by taking advantage of sparsity and by efficiently updating factorizations of the active-set method's indefinite KKT system. In doing so we address the gap between direct and indirect methods. We develop NASOQ to combine the advantages of leveraging direct, accurate solutions of KKT systems with the small and localized updates of subsequent KKT systems. NASOQ leverages our new SoMod method which enables efficient, sparsity preserving updates of existing factorizations after each new constraint update to $\mathcal{W}$ and, as we show in Section 6.5, enables the application of accurate direct factorization methods across a wide range of large- and small-scale QP problems not previously possible.

6.3 SoMod: Sparsity-oriented row modification

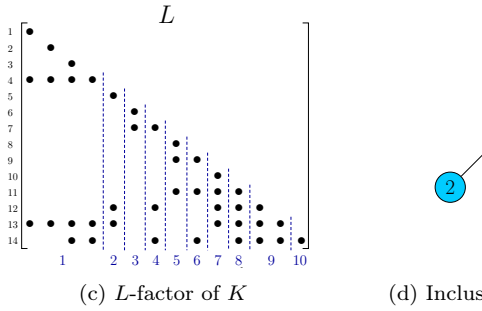
A scalable solution to a dual-feasible active-set QP requires an efficient solution to the successive KKT systems in Equations 6.7 and 6.8. This section discusses SoMod, a novel method for efficiently solving these KKT systems using the combination of a novel sparsity-oriented row modification method, a novel implementation of LDL factorization, and an efficient triangular solve. SoMod consists of two phases: an initialization phase associated with Equation 6.7 and a factor modification phase associated with Equation 6.8. In both phases, SoMod solves $Kx = s$ for x where s is a dense vector and K is a sparse symmetric indefinite KKT matrix. At the start of each QP solve we initialize the KKT matrix with the subsystem corresponding to applying just the equality constraints, so that:

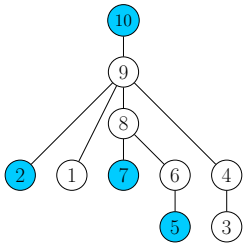
$$K = \begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} \tag{6.9}$$

where H and A are respectively the matrices for the quadratic objective and equality constraints. In order to solve the system $Kx = s$, SoMod applies LDL factorization



(a) Sparse matrices in Equation 6.1 (b) Inclusive matrix (H_{inc})





(c) L-factor of K (d) Inclusive assembly tree (π_{inc})

Figure 6.1: The symbolic initialization phase of SoMod starts with creating an inclusive matrix, shown in Figure 6.1b from the matrices in Figure 6.1a which are inputs to the QP problem in Equation 6.1. The inclusive matrix is then permuted with a fill-reducing permutation to compute the sparsity pattern of the L-factor with minimum number of fill-ins. The sparsity pattern of the L-factor of the inclusive matrix in Figure 6.1b is computed and shown in Figure 6.1c. Boundaries of Supernodes are shown with dotted lines and supernode numbers are illustrated below the L-factor. The corresponding inclusive (assembly) tree of the L-factor in Figure 6.1c is shown in Figure 6.1d. The colored nodes correspond to the inequality constraint rows (matrix C in Figure 6.1a). The constraint-aware supernode creation strategy ensures that supernodes corresponding to the inequality constraint nodes contain only a single column. The colored nodes of the inclusive tree are removed to create the pruned inclusive tree passed to numerical factorization along with the L-factor in Figure 6.1c.

to decompose the matrix K into

$$K = P_{fill} P_S (LDL^T + E) P_S^T P_{fill}^T \tag{6.10}$$

where D is a blocked diagonal symmetric matrix (due to our use of Bunch-Kaufman pivoting [154]), L is a sparse lower triangular matrix, E is the error due to a diagonal perturbation matrix added to K(necessary to avoid zero diagonals, which can cause instabilities [154, 92]), P_{fill} is a fill-reducing ordering (such as METIS [101]), and P_S is reordering due to pivoting. Given this factorization of the matrix, SoMod then uses L and D along with s (the right-hand side) to quickly compute the solution x

via triangular solve.

The overall process of the factorization in this initialization phase of SoMod closely follows that of standard sparse linear system solvers. For efficient factorization, the sparsity pattern of K is analyzed during *symbolic analysis*. Symbolic analysis uses the sparsity pattern of K to construct *symbolic information*, which consists of the fill-reducing ordering P_{fill} and the sparsity pattern of L. Symbolic information guides the *numeric factorization*, which operates on the actual numeric values of K to compute the nonzero values of L and D. Unlike prior work, SoMod applies symbolic analysis in a way that allows the results to be reused during the modification phase. The initialization phase also includes *permutation* with P_{fill}, constraint-aware super-node creation, *perturbation* with E, and a restricted *pivoting* strategy with P_S; all of these steps are described in Section 6.3.1.

The modification phase in SoMod, described in Section 6.3.2, iteratively solves each new, updated KKT system which contains additional active constraints. During this phase, SoMod solves $Kx = s$ for each updated K:

$$K = \begin{bmatrix} H & A^T & C_{\mathcal{W}}^T \\ A & 0 & 0 \\ C_{\mathcal{W}} & 0 & 0. \end{bmatrix} \tag{6.11}$$

Here, for each update, C_w contains rows from the full constraint matrix corresponding to the current proposed active constraint set. Rather than solving each of these systems from scratch, SoMod updates the starting solution in our initialization phase using factor modification. Specifically, SoMod updates the symbolic information from the initialization phase, followed by updating the numeric L-factor to account for the added/removed constraints in each iteration. Then, a triangular solve is once again used to find x given the updated L and D matrices.

6.3.1 Initialization Phase

In SoMod, the initialization phase produces symbolic information that can be reused by subsequent factorizations in the factor modification phase. After producing symbolic information, this phase then proceeds with numeric factorization, followed by triangular solve to return the solution to the KKT system.

Symbolic Analysis with the Inclusive Matrix

The initialization phase first builds an *inclusive matrix*. Then, we permute the inclusive matrix and generate symbolic information, including the sparsity pattern of

the L-factor of the inclusive matrix, the assembly tree of the inclusive matrix (the *inclusive tree*), a pruned inclusive tree, and P_{fill} in Equation 6.10. This symbolic information collectively facilitates an efficient numeric factorization in the initialization phase and also provides symbolic information leveraged by the factor modification stage.

The inclusive matrix, the sparsity of L, and the assembly tree An inclusive matrix is first assembled using the objective and equality constraint matrices (H and A) and the sparsity pattern of the inequality constraint matrix. That is, the inclusive matrix includes all entries of C but with values set to zero. The numerical values of the inequality constraint matrix will be added to the inclusive matrix during the modification stage of SoMod when a constraint is added. Figure 6.1b shows an example of an inclusive matrix created from the matrices in Figure 6.1a.

SoMod then builds an *elimination tree* [116, 37] for the inclusive matrix, which enables obtaining the sparsity pattern of the L-factor and creating the inclusive assembly tree. The elimination tree of the inclusive matrix is a tree that expresses dependencies between operations on columns of the L-factor, dictating the order of factorization. Because of fill-ins, the sparsity pattern of the L-factor is different from that of the inclusive matrix. The number of fill-ins correlates with the number of operations in the factorization process. Thus, after creating the inclusive matrix, it will be permuted with P_{fill}, a fill-reducing ordering, which improves the speed of numeric factorization by reducing the number of operations in the factorization process.

Finally, the inclusive elimination tree and the sparsity pattern of its L-factor are used to create constraint-aware supernodes and the inclusive assembly tree, which is the supernodal version of the inclusive elimination tree. Supernodes [37, 154] are created by grouping columns with similar sparsity patterns if they form a chain in the elimination tree; that is, two consecutive columns are grouped together if one column is the only child of its next column. Thus each node in the assembly tree represents a group of columns together in a supernode; a node in the elimination tree represents just a single column. For example, the tree in Figure 6.1d is the assembly tree of the supernodal L-factor in Figure 6.1c; each node of the tree corresponds to a supernode and the numbers shown below the L-factor correspond to the supernode's number in the assembly tree. The parent of each node in the assembly tree is obtained using the row index of the first off-diagonal nonzero of its corresponding supernode in the L-factor. For example in Figure 6.1c, the row index of the first off-diagonal nonzero of supernode 3 is 4 and thus node 4 is the parent of node 3 in Figure 6.1d.

Sparse factorization can be more efficient when operating on supernodes instead of individual columns [26]. Supernode creation in SoMod is constraint-aware, so

rows/columns of C are not grouped with each other or with other columns; this makes it possible to add or remove constraints separately from one another while still allowing the rest of the assembly tree to benefit from the increased efficiency of the supernodal approach. For example, in the inclusive tree in Figure 6.1d, column 14 can form a supernode with columns 12 and 13; however, since column 14 is the fourth constraint in C, it is excluded from the supernode.

The pruned inclusive assembly tree The inclusive assembly tree contains dummy entries for all inequality constraints. During this phase, and during row modification, we instead use a *pruned* inclusive assembly tree that contains only the entries corresponding to active inequality constraints; this is represented by an array of parents, denoted with π. Thus, before performing the initial factorization, we remove all dummy entries corresponding to inequality constraints. As an additional optimization, SoMod also creates a visibility vector v that shows whether a column of the L-factor should be visited during the initial numerical factorization phase; this information is derived from the pruned assembly tree, but in practice using the visibility vector can be faster than finding paths in the assembly tree. Because the initial KKT matrix only includes equality constraints, the visibility vector is initialized by setting all rows of the inclusive matrix that correspond to rows of C to invisible.

Constraint-aware supernode creation facilitates creating the pruned inclusive tree by ensuring every row of matrix C corresponds to one node in the inclusive tree. This allows removing rows by only changing the pruned inclusive tree as described in Section 6.3.2.

Numeric Factorization with LBL

Numeric factorization computes the nonzero values of the L-factor in solving Equation 6.7 (line 2 of Algorithm 4) using LBL, a modified LDL factorization algorithm that uses Load-Balanced Level Coarsening [28], a scheduling technique that improves the performance of numeric factorization on parallel architectures. While prior work applied Load-Balanced Level Coarsening to Cholesky factorization for symmetric positive definite matrices, LBL extends the technique to symmetric indefinite matrices that arise from KKT problems.

Numeric factorization takes as input the sparsity pattern of the L-factor and the visibility vector, and first computes the *perturbation matrix* (E in Equation 6.10), using information from the inclusive matrix to enable a stable factorization. Perturbation ensures no zeros exist in the diagonal entries of the matrix, since these lead to division-by-zero during factorization. Numeric factorization then uses the pruned inclusive assembly tree to determine an efficient and correct order of computation,

Algorithm 5: Blocked-diagonal LDL factorization. Matrices K, L, D, P_S correspond to Equation 6.10. *super* is a vector that shows the boundary of supernodes in L. $\mathcal{M}$ is a set that shows the order of computation. Matrices L and D only have the sparsity pattern for LBL and include the previous factor for row modification.

Data: K, L, D, *super*, $\mathcal{M}$
Result: L, D, P_S
for $j \in \mathcal{M}$ **do**
  $b = super_j$
  $u = super_{j+1}$
  $L_{:,b:u} = 0$
  $T(:,:) = 0$
  /* Applying contributions from factorized supernodes in r */
  for $r \in L_{0:b,:}$ **do**
    $T = T + L_{b:n,r} \times D_{r,r} \times L^T_{b:u,r}$
  end
  $[L_{b:u,b:u}, D_{b:u,b:u}, P_{Sb:u,b:u}] = \text{LDL}(K_{b:u,b:u} - T_{b:u,:})$
  /* Applying column permutation */
  $L_{:,b:u} = L_{:,b:u} \times P^T_{S\,b:u,b:u}$
  $L_{u:n,b:u} = (L_{b:u,b:u} \times D_{b:u,b:u})^{-1} \times (K_{u:n,b:u} - T_{u:n,:})$
end
/* Applying row permutation */
$L = P_S \times L$

and then uses this schedule to compute the nonzero values of L, D, and P_S in Equation 6.10.

Perturbation We add a small value to zero diagonals of the inclusive matrix that correspond to rows of the equality constraints. Since the location of the equality constraints are known in the inclusive matrix, SoMod computes the perturbation matrix E:

$$E_{i,i} = diag\_pert; \quad n \leq i \leq n + m \tag{6.12}$$

where n and m are the number of variables and constraints respectively. Matrix E will be added to the inclusive matrix as shown in Equation 6.10.

Load-Balanced Level Coarsened scheduling Before performing the factorization, we use Load-Balanced Level Coarsening to compute the order of factorization using the pruned inclusive assembly tree. This scheduling algorithm provides a partitioning of the tree that groups supernodes into partitions that can execute efficiently on a parallel processor while preserving ordering dependencies. For example, in the pruned tree of Figure 6.1d, nodes 1, 3, and 6 can run in parallel, since none of them depend on lower non-colored nodes in the tree.

LBL: parallel blocked-diagonal LDL factorization LBL is a parallel LDL factorization method that takes the computed schedule from the Load-Balanced Level Coars-

ening algorithm, the visibility vector, and the sparsity pattern of the L-factor and computes the nonzero values of the L and D factors of the perturbed KKT matrix of the system in Equation 6.7 (line 2 in Algorithm 4). Pseudocode for LBL is shown in Algorithm 5.

LBL uses a supernodal left-looking approach [37] (one of several ways to compute LDL factorization), computing the supernodes of the L-factor using already factorized supernodes to the left of the current supernode. The list of supernodes and the order of computation are specified in *super* and $\mathcal{M}$ respectively. Each iteration of LBL first accumulates contributions of supernodes to the left and stores them in temporary matrix T. After deducting T from the current column (line 9), the algorithm first factorizes the diagonal part of the current supernode using a dense LDL factorization and then uses the computed factors to factorize the off-diagonal part of the current column (line 11). The dense LDL factorization uses the Bunch-Kaufman algorithm, which only reorders rows within a supernode of the L-factor. Thus, LBL pivoting is restricted to rows within a supernode [154], which preserves the sparsity pattern of the L-factor during factorization.

After pivoting, rows of the L-factor in supernodes to the left of the current supernode must be permuted as well; were this done within the parallel region (lines 2–11), it would introduce dependencies that would prevent efficient computation, rendering the Load-Balanced Level Coarsening schedule useless. Thus, unlike typical LDL factorization methods, LBL separates row and column permutations, applying row permutations after the factorization (line 13). After obtaining the factorization, SoMod then uses triangular solve to efficiently obtain a solution to the initial KKT problem, as described in Section 6.3.3.

LBL thus works with the same base SBK algorithm as in MKL Pardiso [154]. However, LBL enables additional important features necessary for updates. The first and most key feature is that LBL enables factor modification: when adding or removing a constraint from K, LBL modifies the factor as opposed to MKL Pardiso which requires computing the factor from scratch. The second feature is LBL's application of a static scheduler [28] to schedule the computation to ensure load-balanced parallelism while preserving locality. In contrast, MKL Pardiso utilizes dynamic scheduling which optimizes solely for load-balanced execution, which results in suboptimal locality. To prevent dependencies due to pivoting in SBK that would limit parallelism, LBL postpones row permutation to after numerical factorization.

Algorithm 6: Symbolic row removal algorithm. k is the node to remove. π is the pruned inclusive assembly tree. v is the visibility vector. r is the list of root nodes. $\pi^{-1}(k)$ returns the children list of node k.

```
   Data: π, v, k, r
   Result: π, v, r
   /* Find the parent of deleting node k                                  */
 1 f = π(k)
   /* Update all children of node k with its parent                      */
 2 for j ∈ π^{-1}(k) do
 3 |   if v(j) then
 4 |   |   π(j) = f
 5 |   |   if k is a root node then
 6 |   |   |   r = r ⋃ {j}
 7 |   |   end
 8 |   end
 9 end
10 r = r − {k}
   /* Update the visibility vector                                       */
11 v(k)=false
```

6.3.2 Factor Modification

After the initialization phase, finding a solution to the QP problem requires solving a large number of successive symmetric indefinite KKT systems. The factor modification phase in SoMod efficiently solves these successive systems by reusing the computed factors from the initialization phase and modifying them based on whether a new inequality constraint is added or removed. In contrast, the usual approach would solve these systems from scratch, performing symbolic analysis and factorization without reusing any previously-computed information.

Successive KKT matrices are created by adding or removing rows of matrix C to/from the existing KKT system. To obtain the solution to the linear system in Equation 6.8 (line 4 of Algorithm 4), SoMod first updates the symbolic information and then the numeric factorization previously obtained from the initialization phase or obtained from the previous iteration of the QP algorithm. It then uses the updated L-factor and D to obtain a solution (Section 6.3.3). In this subsection we explain how factor modification efficiently modifies the previously obtained symbolic information and then uses the new symbolic information to update the existing numeric factors.

Symbolic Modification

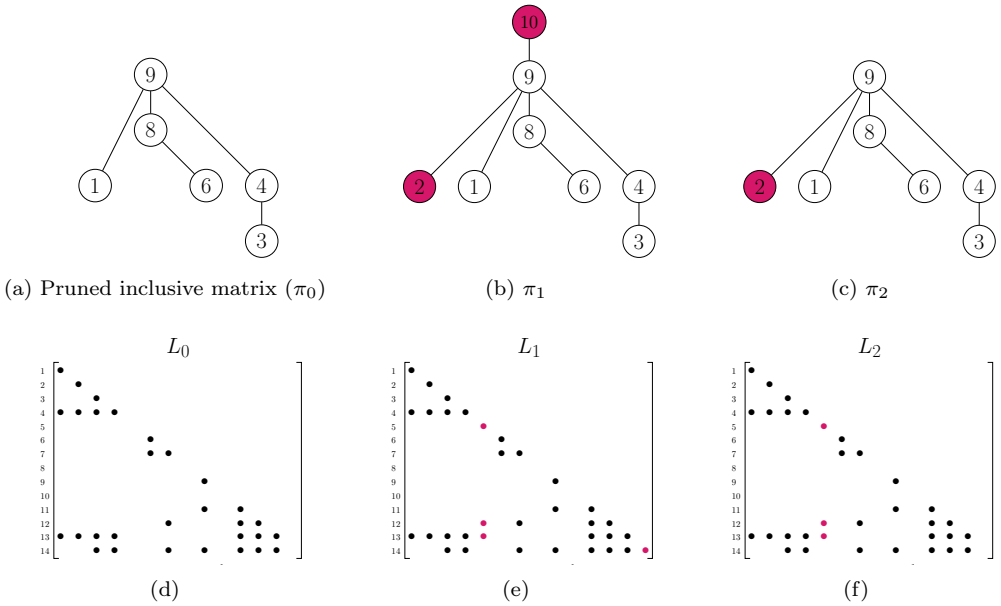
The symbolic modification phase modifies the pruned inclusive assembly tree using the full inclusive tree when row k of the inclusive matrix is modified. Depending on whether the modification adds or removes a row, SoMod uses symbolic row removal or addition algorithms to update the tree.

Algorithm 7: Symbolic row addition algorithm. k is the node added. π is the pruned inclusive assembly tree. π_{inc} is the inclusive assembly tree. v is the visibility vector. r is the list of root nodes. $\rho_{inc}(j)$ returns the list of ancestors of node j. $\pi^{-1}(f)$ returns the children list of node f.

Data: π, π_{inc}, v, r, k
Result: π, v, r
/* Find the first visible ancestor of k */
$f = \min \{j | j \in \rho_{inc}(k) \wedge v(j)\}$
if *f is a node* **then**
  /* Find all nodes that k is their least ancestor */
  for $j \in \pi^{-1}(f)$ **do**
    if $k \in \rho_{inc}(j)$ **then**
      $\pi(j) = k$
    end
  end
else
  /* Look for any missing child in root nodes */
  for $j \in r$ **do**
    if $k \in \rho_{inc}(j)$ **then**
      $\pi(j) = k$
      $r = r$ - $\{j\}$
    end
  end
  $r = r \bigcup k$
end
/* Update the pruned inclusive assembly tree */
$\pi(k) = f$
$v(k)$=true

Row removal When a constraint is removed from the KKT matrix, SoMod updates the pruned inclusive tree using the symbolic row removal algorithm shown in Algorithm 6. To remove node k, the removal algorithm first finds its parent and then assigns all children of node k to its parent. If node k is a root node and therefore has no parent, we add its children to a list r which contains all root nodes. This list is used in the row addition algorithm to facilitate the process of adding constraints corresponding to such nodes. For example, Figure 6.2b shows a pruned inclusive assembly tree with two already-added constraint rows 2 and 10 and Figure 6.2c shows the pruned inclusive assembly tree after node 10 is removed. Node 9 then becomes a root node, so Algorithm 6 adds it to r and removes 10 from r.

Row addition When row k is added to the KKT matrix, SoMod updates its symbolic information and decides where to insert node k in the pruned inclusive assembly tree. The algorithm first visits the tree to find the first visible ancestor of k. If there is a first visible ancestor, the algorithm then finds the children of this closest ancestor f in the pruned inclusive tree; the children are then visited to update any for which k is the parent. If k does not have a first visible ancestor, the algorithm cannot use



(a) Pruned inclusive matrix (π_0) (b) π_1 (c) π_2

(d) (e) (f)

Figure 6.2: Factor modification example starting with the pruned inclusive tree (Figure 6.2a) and the L-factor (Figure 6.2d) that are computed in the initialization phase, in order, by removing all nodes corresponding to the inequality matrix from the inclusive tree in Figure 6.1d and by hiding all rows of the inequality matrix in the L-factor in Figure 6.1c. SoMod symbolically adds rows that correspond to nodes 2 and 10 (rows 5 and 14, respectively) to the inclusive matrix using the row addition algorithm, resulting in a new pruned inclusive tree shown in Figure 6.2b. The corresponding supernodes in the L-factor in Figure 6.2e, shown in red, are also visible and will be updated using the numerical modification algorithm. Figure 6.2c is the result of removing node 10 from Figure 6.2b by using the symbolic row removal algorithm. Column 14 of the L-factor (which corresponds to node 10 in the tree) in Figure 6.2f becomes invisible after row removal.

its ancestor's information to update the pruned inclusive tree and thus would need to search for its children by considering all nodes of the inclusive tree. Instead, the algorithm uses the list of root nodes from the node removal algorithm and only searches in r. Algorithm 7 demonstrates the process of row addition. Figure 6.2a shows the pruned inclusive tree with no constraints and Figure 6.2b shows the pruned inclusive assembly tree after adding constraint rows 2 and 10. When adding node 2, Algorithm 7 first finds its visible ancestor, node 9, and then updates the parent node with 9 since no child of node 9 belongs to node 2. Node 10 is a root node in the inclusive tree in Figure 6.1d, thus the algorithm goes over the list of root nodes, which includes 9 as explained in the example in the row removal section. Since 9 is a child of 10 in the inclusive tree, its parent will be updated with 10.

Numeric Modification

SoMod uses the updated pruned inclusive tree to update the numeric factorization of the newly-modified KKT system by visiting only the columns that are dependent on the modified row in the pruned inclusive assembly tree. Given the pruned inclusive assembly tree of a linear system, solving the factorization after adding or removing a row is similar to Algorithm 5. The only difference is how the input $\mathcal{M}$ is computed. Numeric modification only updates supernodes that are in $\rho(k)$, which are the nodes in an *up-traversal* starting from k. An up-traversal from a node visits all ancestors of that node. For example, the up-traversal of node 2 in the tree of Figure 6.1d is {2, 9, 10}. For the node removal case, the symbolic row removal algorithm is called after the numeric modification. This allows the modification algorithm to apply the effect of removing the node before removing the necessary information from the symbolic information; instead, the row is replaced with all zeros in order to update the numeric factorization. For the node addition case, the symbolic row addition algorithm is called before numeric modification, and the appropriate nonzeros are added to the row. Calling Algorithm 7 before numeric modification allows the numeric modification algorithm to utilize values of the added row during the L-factor update.

6.3.3 Triangular Solve and Accuracy Refinement

In both the initialization phase as well as the modification phase, once SoMod obtains the newly-computed or updated numeric factors L and D, it uses them to solve the linear system $Kx = s$. SoMod finds the solution vector x by doing a forward triangular solve, a block-diagonal system solve, and a backward triangular solve as shown in Equation 6.13. SoMod uses an efficient parallel triangular solve from [28] and a simple single-threaded hand-written block-diagonal system solver for these steps.

$$\begin{aligned} b &= (P_{fill}P_S)s \\ Lx_1 &= b \\ Dx_2 &= x_1 \\ L^T x_3 &= x_2 \\ x &= (P_{fill}P_S)^{-1}x_3 \end{aligned} \tag{6.13}$$

As shown in Equation 6.10 and discussed in Section 6.3.1, we add perturbation matrix E to the KKT matrix prior to solving. As a result, the solution x contains inaccuracies, necessitating an accuracy refinement strategy to obtain an acceptable solution.

Algorithm 8: NASOQ: A dual-feasible full-space QP solver.

Data: H, q, A, b, C, d
Result: x, y, z
/* Feasibility phase */
LinearSolve SoMod(H, A, C);
$z_0 = 0$; $k = 0$; active-set=$\emptyset$; modify=ADD;
SoMod.symbolic_initialization();
SoMod.LBL(q, b);
$[x_0, y_0]$ = SoMod.solve();
/* Optimality phase */
while x_k *is not primal-feasible* **do**
  w = most_violated(C, d, x_k);
  if *modify* == *ADD* **then**
    SoMod.symbolic_row_addition(w);
    SoMod.numerical_modification(w);
  else
    SoMod.numerical_modification(g);
    SoMod.symbolic_row_removal(g);
  end
  $[\Delta x, \Delta y, \Delta z]$=SoMod.solve();
  Compute the step length t;
  if $t = \infty$ **then**
    Problem is unbounded.;
  else
    Update $x_{k+1}, y_{k+1}, z_{k+1}$ with $\Delta x, \Delta y, \Delta z$;
    Update the active set;
    Add or remove a row to/from KKT;
    Set g to removed constraint;
    Set $modify$ to either `ADD` or `REMOVE`;
  end
  k = k + 1 ;
end
x_k, y_k, z_k are optimal;

Accuracy refinement It is standard practice to use an iterative method after a direct solve to improve the accuracy of the solution [86, 156, 152, 11]. SoMod uses an iterative method, right-preconditioned GMRES [152], to refine the obtained solution from the solve phase. The GMRES algorithm is preconditioned with the output of LDL factorization and performs up to *max_iter* iterations to achieve the requested residual norm *res_tol*, but will terminate early if the required tolerance is achieved.

6.4 NASOQ: Numerically Accurate Sparsity-Oriented QP Solver

With our key innovation SoMod in place, we now can define our two closely-related QP solution algorithms, NASOQ-Fixed and NASOQ-Tuned. Both methods integrate SoMod row modification and LBL within the GI dual-feasible active-set framework

and so provide efficient, accurate, and sparsity-preserving full-space QP algorithms. NASOQ-Fixed and NASOQ-Tuned both, as we show in Section 6.5, consistently improve over state-of-the-art QP methods across our benchmark, while the two methods each individually offer a different balance in the trade-off between efficiency and accuracy for larger scale problems.

In this section we first highlight the changes applied by SoMod row modification and LBL to the GI framework with NASOQ-Fixed and then, building off of this baseline, discuss the NASOQ-Tuned method as a direct and natural extension of NASOQ-Fixed.

Algorithm 8 summarizes our full NASOQ-Fixed algorithm in pseudocode. At each update and solution of the new active-set KKT (previously lines 2 and 4 in Algorithm 1) NASOQ-Fixed now applies the SoMod solve phase via LBL and row modification. This allows NASOQ-Fixed to replace the Cholesky and QR solves in the standard GI method; this is the key difference between NASOQ and standard GI methods. In addition, standard GI implementations require one additional iteration for each equality constraint while NASOQ, due to its full-space approach, applies equality constraints by solving the initial KKT system, see Equation (6.7).

Numerical optimization methods generally apply a wide diversity of empirically tuned parameters [170, 133]. A key feature of NASOQ is that in our construction of SoMod's LBL and row modification we expose three parameters with direct and intuitive interpretations that enable us to balance efficiency against accuracy for different applications and problem scales. With NASOQ-Fixed we demonstrate that without tuning a default setting works well across the board. With NASOQ-Tuned we similarly demonstrate that if a range of reasonable settings for these parameters are a priori known, NASOQ's active set approach enables a rapid sweep for improved accuracy. These parameters are

- *max_iter*: the maximum number of refinement iterations for incrementally improving the solution of a KKT system after the solve phase (see Section 6.3.3);
- *stop_tol*: the threshold defining the upper bound for the residual accuracy of the KKT system during the refinement phase (see Section 6.3.3);
- *diag_perturb*: value added to zero-entry diagonals of the KKT matrix (see Section 6.3.1) to stabilize LBL and row modification in SoMod.

Increasing *max_iter* and decreasing *stop_tol* generally improve accuracy at the cost of more computation. Setting of *diag_perturb* then relates to problem conditioning and machine accuracy; a typical value is near square root of machine precision [11].

6.4.1 NASOQ-Fixed

NASOQ-Fixed is a direct, one-shot application of our SoMod-enhanced GI approach. Here we presume that, per-application, time is sufficient for a single QP solve, but no further, and so seek the most effective values for given input QP characteristics.

NASOQ-Fixed sets *diag_perturb* and *stop_tol* to fixed values *for all* input QP problems to 10^{-9} and 10^{-15} respectively. Here we find that adapting *max_iter* per problem based on the requested accuracy ϵ is most effective (with the assumption that we will only have a single attempt at the solve). When lower accuracy ϵ's are requested, NASOQ-Fixed applies fewer refinement iterations (lowering *max_iter*) and then correspondingly increases *max_iter* to obtain more accurate linear solves when higher accuracies are specified. Concretely, NASOQ-Fixed sets

$$max\_iter = \begin{cases} 1, & \epsilon > 10^{-4} \\ 2, & 10^{-8} \leq \epsilon \leq 10^{-4} \\ 3, & \epsilon < 10^{-8}. \end{cases} \tag{6.14}$$

Finally, for very small QP problems, i.e., fewer than 100 nonzeros, NASOQ-Fixed keeps $max\_iter = 3$ as this does not impose an appreciable cost.

6.4.2 NASOQ-Tuned

NASOQ-Tuned leverages the underlying active-set framework. Active-set methods terminate in bounded time with respect to number of constraints, and, in practice generally much more rapidly than barrier and first-order approaches [197, 120]. Thus the cost of running multiple passes of NASOQ to determine whether a chosen setting for our three parameters successfully matches a requested accuracy is generally acceptable when accuracy is critical and some efficiency can be sacrificed.

NASOQ-Tuned therefore sweeps through a set of empirically-determined parameter combinations, as found from testing solely against the subset (0.5%) of the QP problem instances in the testing set (see Section 6.5.3) for which NASOQ-Fixed does not converge. NASOQ-Tuned begins with an initial pass of NASOQ-Fixed. Then, if the requested accuracy ϵ is not met, it successively tries new NASOQ passes with the sequence of configurations in Table 6.1.

In practice, for all but 21 examples in our benchmark, we find that NASOQ-Tuned successfully converges at all requested accuracies. See Section 6.5.3 for details.

Table 6.1: List of NASOQ-Tuned parameters. Each row contains parameters used in one pass of NASOQ-Tuned.

Config	`max_iter`	`diag_perturb`	`stop_tol`
1	2	10^{-9}	10^{-15}
2	2	10^{-13}	10^{-15}
3	2	10^{-7}	10^{-15}
4	2	10^{-11}	10^{-17}
5	3	10^{-10}	10^{-15}
6	3	10^{-9}	10^{-17}
7	3	10^{-11}	10^{-17}

6.5 Experimental Results

In this section we evaluate NASOQ against other solvers on a large set of QP problems from diverse applications. First, we describe our experimental setup (Section 6.5.1) and our new collection of 1513 sparse QP problems collected from a wide variety of applications (Section 6.5.2). We then evaluate NASOQ for the full benchmark set, comparing accuracy, efficiency, and scalability against existing tools (Section 6.5.3). We then describe the effect of numerical range on NASOQ's ability to attain convergence (Section 6.5.4). Finally, we explore the impact of SoMod on overall performance (Section 6.5.5).

Across all QP problems in our repository, NASOQ converges for over 99.5% of the problems for accuracies ranging from 10^{-3} to 10^{-9} with average speedups ranging from 1.7× to 24.8× over the best competing method. For requested accuracies of 10^{-3} and 10^{-6}, NASOQ-Tuned has no failures, while only 21 problem instances (out of 1513, or 1.4%) fail to reach the 10^{-9} accuracy threshold. Our analysis shows that these few failures occur due to the numerical range of the problems themselves and that other solvers likewise fail to solve these problems. NASOQ demonstrates consistent efficiency and speedups across all application types, and we see that the SoMod algorithm plays a critical role in the performance of NASOQ.

6.5.1 Experimental Setup

Testbed architecture All experiments are performed on a 6-core Haswell-E described in Table 2.1 with turbo-boost disabled, running Ubuntu 16.04 with Linux kernel 4.4.0. NASOQ and all open-source solvers are compiled with GCC v5.4.0 using the `-O3` option. MKL 2019.1.144 is used wherever dense BLAS routines are required. Throughout this section, *convergence time* refers to the wall clock time to reach convergence, measured using the standard C++ `chrono` library. We use a time limit of 30 minutes for all solvers; if a solver does not converge in 30 minutes for a problem

instance, we consider it a failure for that instance.

Termination criteria We set all solvers, where possible, to use common, absolute (rather than relative) termination criteria, i.e., a common accuracy threshold ϵ for all four measures in Equations 6.3-6.6. Relative tolerances are often specific to the algorithm and/or particular domain, and are often highly susceptible to falsely reporting convergence when an algorithm stagnates (e.g. small relative errors only tell us iterates have stopped progressing) rather than reaching a low error solution. Although accuracies for each of the four optimality measures in Equations 6.3-6.6 change depending on application, we believe a desirable goal for a general-purpose QP solver is to solve every reasonable problem to any requested accuracy given commensurate time, and to only report success when accuracy is achieved. We don't presume to know a priori per problem type what measures are most important; instead, we evaluate fitness by asking each method to drive all measures down below each specified error tolerance according to the infinity norms of Equations 6.3-6.6. Details for each solver are explained separately below.

Solver settings We compare NASOQ with four widely-used state-of-the-art QP solvers: OSQP [170], Gurobi [134], MOSEK [129], and QL [159]. These tools are selected to represent different QP solver methods. OSQP applies a first-order method, supports sparse problems, and parallelism. Gurobi and MOSEK are both commercial tools based on barrier methods; both support parallel execution and sparse QP problems. To compare NASOQ to an alternative Goldfarb-Idnani algorithm implementation, we include QL, a robust GI implementation; however, QL does not support sparsity nor parallelism.

NASOQ is implemented in C++ with double precision, with METIS 5.1.0 for reordering the inclusive matrix, and MKL BLAS [192] for dense operations within LBL. All other QP solvers are set to their default modes and only settings related to the requested accuracy or ϵ in Equations 6.3-6.6 are changed when exposed and necessary, see below[4].

OSQP is an open-source first-order solver designed for sparse QP problems. We use OSQP 0.6.0 and build in double precision using the MKL Pardiso solver. In OSQP the user-requested accuracy is scaled by the norm of matrix H (see [170] Section 3.4). This scaling leads to early termination and thus inaccurate/non-optimal solutions. For fair comparison, we change OSQP's termination criteria to use the absolute requested accuracy – leaving the algorithm otherwise unchanged[5].

Gurobi and MOSEK are two commercial solvers that apply barrier methods to solve QP problems. For both packages, we utilize default settings for the solvers. We

[4]The list of parameters related to user-requested accuracy for each solver is provided in Section A.3.
[5]Detailed information on the sole parts of the OSQP code we modify is provided in Section A.3.

use currently latest releases of MOSEK (v8) and Gurobi (v8). Gurobi uses an absolute termination criteria and so can be applied in our comparison directly. MOSEK, on the other hand, does not allow absolute error tolerances for convergence and instead applies algorithm-specific, relative measures. As MOSEK is closed source (and so its termination criteria can not be modified) we experimented with a range of its different exposed parameter settings, seeking to maximize MOSEK's success. Discussion of our experiments with MOSEK and details of MOSEK's behavior applied with its most successful settings for comparison are covered in Section 6.5.3 below.

QL is a dense active-set solver based on the GI algorithm, implemented in Fortran. We convert all sparse matrices to dense prior to using QL, as it only supports dense matrices. Conversion time is not included in reported solve times. Large-scale QP problems cannot be converted due to memory limitations of the testbed architecture.

Performance profile Aggregating combined performance and failure data in plots across methods on a significantly-sized benchmark is always challenging. Thus, to compare the convergence speed of different solvers, following existing work [46, 196, 170, 135] we utilize a performance profile plot. To define performance profiles, we use the performance ratio $r_{p,s} = \frac{t_{p,s}}{min_s t_{p,s}}$ where $t_{p,s}$ is the time for QP solver s to solve problem instance p. When solver s fails for problem p, its performance ratio is set to infinity, i.e., $r_{p,s} = \infty$. After the performance ratio for all pairs of solvers and problem instances is obtained, we compute the performance profile, function f_s, that maps any $r_{p,s}$ to $[0, 1]$ and is computed as: $f_s(\tau) = \frac{1}{n_p} \sum_p \alpha_{\leq \tau}(r_{p,s})$ where $\alpha_{\leq \tau} = 1$ if $r_{p,s} \leq \tau$ and n_p is the number of problems in our repository. $f_s(\tau)$ denotes the fraction of solved problems within $\tau\times$ the time of the best solver. Thus, in Figure 6.4 for example, faster performance for a given fraction of problems means the line is to the left, while more problems with successful convergence lead to lines that are higher on the y-axis.

Speedup In addition to performance profiles, we also provide detailed, per-category analyses and breakdowns using speedup and failure rate (Section 6.5.3). The reported average speedup throughout the chapter is computed using normalized shifted geometric mean [170, 125]. Given $t_{p,s}$ is the time for QP solver s to solve problem instance p, shifted geometric mean of solver s across n problems is computed as: $g_s = \sqrt[n]{\prod_p (t_{p,s} + k)} - k$ where k is the shift and selected to be one [170]. When solver s fails in the problem p, $t_{p,s} = 1800$ which is the 30 minute time limit in seconds. To avoid overflow we use the logarithmic form of the geometric mean. Given g_s for each solver, the speedup of solver $s1$ over $s2$ is computed by $\frac{g_{s2}}{g_{s1}}$.

6.5.2 Benchmark Repository for Sparse Quadratic Programs

We assemble a repository for sparse QP problems of different scales, most of which come from applications in animation, geometry processing, and simulation. Existing QP problem benchmarks are not large enough to stress-test large-scale QP solvers. For example, the largest QP problem instance in terms of the number of variables in the Maros-Mészáros repository [121] only has 10k variables, which is far smaller than real-world large-scale QP problems. Existing QP solvers are either tested for a limited number of problems or are tested for randomly generated problems [170, 135]. To address this shortcoming, we gathered existing strictly-convex QP benchmark problems and also added a set of new QP problem instances mostly arising from computer graphics applications.

Our repository includes QP instances from shape deformation, contact simulation, model reconstruction, and cloth simulation from computer graphics; model predictive control (MPC) [160] from robotics; and strictly-convex QP problems from the Maros-Mészáros repository [121]. The number of variables ranges from 50–114309 and the number of constraints ranges from 20–10k. Each QP for image deformation comes from Jacobson *et. al.* [95] and is created using libigl [96]. Contact simulation QPs correspond to QP problems that must be solved in each timestep of the simulation and are created using the GAUSS library [109]. Model reconstruction instances are QP problems that compute the third dimension of a 2D mesh, explained in [177, 53]. Cloth simulation QPs arise from each timestep of the cloth simulation in [194].

6.5.3 Accuracy, Efficiency, and Scalability of NASOQ

NASOQ can solve a large range of QP problems from different application types and across a range of problem scales. In this section, we first compare the efficiency and scalability of NASOQ to other QP solvers and demonstrate NASOQ's superior performance. We also explore the performance of NASOQ versus other tools for different types of applications. Finally, we discuss the effect of using the full-space method in NASOQ.

Overall performance

As discussed in Section 5, NASOQ-Fixed and NASOQ-Tuned target different points in the trade-off between efficiency and accuracy. NASOQ-Tuned sweeps through a set of parameters to deliver improved accuracy for problems where accuracy is critical. Thus, as shown in Figure 6.3, NASOQ-Tuned always converges for requested accuracy thresholds of 10^{-3} and 10^{-6}, while NASOQ-Fixed fails for 1.2% of problems

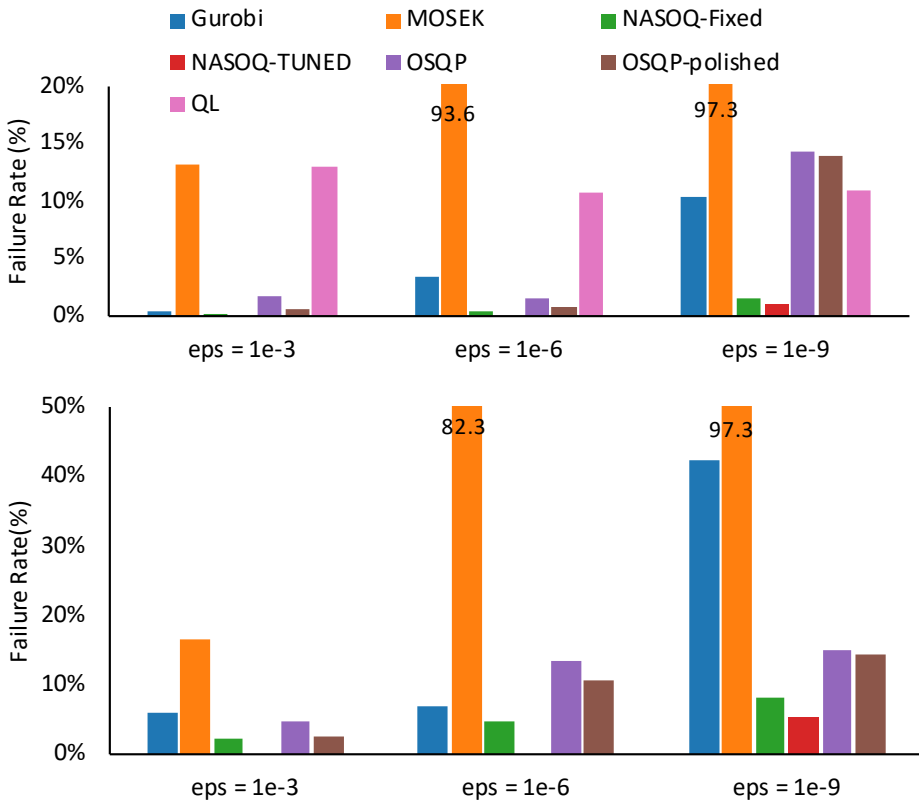


Figure 6.3: Failure rate of NASOQ-Fixed, NASOQ-Tuned, OSQP, OSQP-Polished, Gurobi, QL, and MOSEK across different ranges of accuracy (10^{-3}, 10^{-6}, and 10^{-9}) and for both small-scale (top) and large-scale (bottom) QP problems. NASOQ-tuned has the lowest failure rate compared to all other QP solvers for problems of different scales and for different requested accuracies.

(there are 21 problem instances that NASOQ-Tuned fails for 10^{-9}; this is explained in Section 6.5.4). Since NASOQ-Tuned starts from the NASOQ-Fixed configuration, the performance profiles of both variants are similar, as shown in Figure 6.4 and the small difference is due to problems that NASOQ-Tuned converges and NASOQ-Fixed fails. The convergence behaviour of both variants of NASOQ is consistently better than other solvers for both large- and small-scale problems (Figure 6.3).

OSQP uses several lightweight iterations to incrementally improve the accuracy of the solution to the QP problem. However, when an accurate solution is needed, the number of iterations significantly increases in OSQP, leading to reduced efficiency. Like NASOQ-Tuned, OSQP also has a variant, called OSQP-polished, that trades off efficiency for accuracy in problems where accuracy is critical. OSQP-polished uses an additional step after OSQP to refine accuracy and obtain solutions for some problems when the accuracy range is 10^{-9}. OSQP and OSQP-polished collectively solve 94% percent of all problems in our repository for accuracy ranges of 10^{-3}, 10^{-6}, 10^{-9} which is quite good, but still considerably less than the 99% obtained from NASOQ (see Figure 6.3). NASOQ is more efficient than OSQP across all problem scales and for

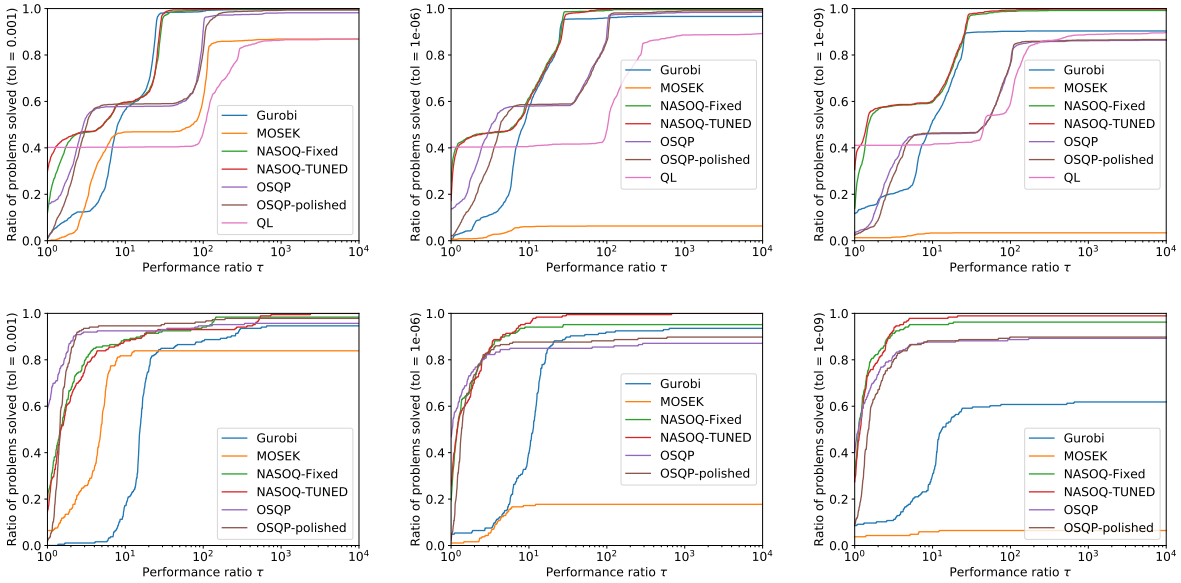


Figure 6.4: Performance profiles for NASOQ-Fixed, NASOQ-Tuned, OSQP, OSQP-Polished, Gurobi, QL, and MOSEK across different ranges of accuracy (from left to right: 10^{-3}, 10^{-6}, and 10^{-9}) and for small-scale (top) and large-scale (bottom) QP problems from our repository. Lines to the left are more efficient, and lines higher on the y-axis solve a greater percentage of problems within a given performance threshold. The figures show that NASOQ-Fixed and NASOQ-Tuned are, for almost all accuracies and all problem scales, more efficient than available QP solvers and are able to solve more of the QP problems in our repository.

different accuracy thresholds. For example, the average speedup of NASOQ-Fixed over OSQP for thresholds of 10^{-3} and 10^{-6} is 2.7× and 2.3× respectively.

Gurobi, in contrast to NASOQ, has a high failure rate and does not scale to larger problems. Unlike NASOQ and OSQP, Gurobi does not provide different variants to balance accuracy and efficiency. In Gurobi, the number of iterations typically remains unchanged for different requested accuracies. Thus, in Figure 6.4, all performance profiles for Gurobi follow similar trends across different requested accuracies and different problem scales. For accuracies of 10^{-3} and 10^{-6}, Gurobi's failure rate is similar to that of OSQP; however, compared to NASOQ, Gurobi fails in more problems. Furthermore, Gurobi exhibits a high failure rate for large-scale problems with lower requested error. For example, for the threshold of 10^{-9}, Gurobi fails for 42.25% of large-scale problems as shown in Figure 6.3.

MOSEK is another barrier method that converges in a bounded number of computationally-heavy iterations. MOSEK does not converge for most large-scale problems with accuracy thresholds lower than 10^{-3}. As shown in Figure 6.3, the failure rate of MOSEK for smaller requested accuracy thresholds is more than 82%, which is significantly higher than the failure rate of all other solvers. As discussed in Section 6.5.1, MOSEK

doesn't allow absolute error tolerances and instead applies algorithm-specific, relative measures. We experimented with a number of different parameter settings, attempting to improve MOSEK's success. During this process we observed that decreasing requested accuracies further below 10^{-10} produces slower performance and increased failures. For example, requesting 10^{-16} accuracy leads failure rates to increase to 86%. We find setting to the requested accuracy works best for MOSEK in terms of combined performance and failure rate reduction. We also set MOSEK's infeasibility tolerance parameter to the default: 10^{-12}. We find no change for high accuracy benchmarks (i.e. 10^{-6} and 10^{-9}) and a 0.2% reduction in failure rate for 10^{-3}. We observe, however, consistent with [170] the speed and failure rate of MOSEK generally lags behind OSQP at low accuracy and Gurobi at higher accuracies.

QL is a dense active-set solver and thus can only solve small-scale problems. For small QP problems, QL's failure rate is 11% for each of the accuracy thresholds as shown in Figure 6.3. The figure also shows that the performance profile and efficiency of QL in Figure 6.4 is inferior compared to other QP solvers including NASOQ, due to its lack of support for sparsity and parallelism.

Effect of Different Applications

Different applications create varying types of QP problems that pose different challenges to solvers. We examine the obtained accuracy and efficiency with different QP solvers as we vary QP problem types. Our analysis shows that unlike other QP solvers, NASOQ performs well across different application domains.

To show this variation, we compare NASOQ-Tuned, NASOQ-Fixed, OSQP, and Gurobi across different application types for the accuracy threshold of 10^{-6}; the trend holds for other accuracies. QL and MOSEK do not successfully converge for larger problem sizes, so we exclude them from our comparison.

For contact simulation problems, NASOQ-Tuned and NASOQ-Fixed provide the lowest failure rates (0% and 0.15%, respectively) compared to all other solvers. Although OSQP's failure rate (1.07%) is higher than NASOQ, it still performs better than Gurobi, which fails for 3.44% of these instances. The efficiency of these solvers also follows the same trend where both NASOQ solvers are faster than OSQP in 80% of contact simulation problems with an average of 2.1× speedup across all contact simulation instances. OSQP also exhibits better efficiency than Gurobi.

In shape deformation and model reconstruction, NASOQ-Fixed and NASOQ-Tuned do not fail for any problems while OSQP and Gurobi fail in 12.5% of instances. NASOQ is 22.8× and 24× faster than OSQP and Gurobi respectively for these problems.

For Maros-Mészáros problems, NASOQ-Tuned does not fail for any problem and the nearest competitors are Gurobi and NASOQ-Fixed with failure rates of 15% and 24.5%, respectively. NASOQ-Tuned is on average 8.9× faster than Gurobi and NASOQ-Fixed is slower than Gurobi by 3.2×. OSQP does not perform well for Maros-Mészáros problems (45% failure rate).

For model predictive control (MPC) problems, NASOQ-Tuned and NASOQ-Fixed show no failures while OSQP's failure rate is 2.5%, which is relatively high compared to Gurobi, which fails in only 0.83% of instances. Both NASOQ-Tuned and NASOQ-Fixed solvers are faster than OSQP and Gurobi. For example, NASOQ-Fixed obtains an average speedup of 3.4× over OSQP-polished.

Unlike other existing solvers, NASOQ provides consistent efficiency and good accuracy across all problem types. Both variants of NASOQ are more efficient and accurate compared to all solvers, with the exception of the failure rate of NASOQ-Fixed for Maros-Mészáros problems [6].

Effect of the Full-Space Approach

As discussed in Section 6.4, NASOQ replaces the range-space method in GI with a full-space approach. In this section we examine the effect of the full-space approach on the accuracy of NASOQ to demonstrate that the use of a full-space method does not negatively affect the accuracy of NASOQ compared to a range-space approach. To show the accuracy of NASOQ's full-space method, we integrate a range-space method inside NASOQ and use it to solve the KKT systems. We call this implementation NASOQ-Range-Space. Cholesky decomposition along with the QR decomposition are used instead of SoMod in NASOQ-Range-Space. However, due to the use of QR decomposition that has intensive memory usage, NASOQ-Range-Space is limited to solving small-scale problem instances. Table 6.2 shows the failure rates of NASOQ-fixed, NASOQ-Tuned, and NASOQ-Range-Space for small-scale problems in our QP dataset. NASOQ-Fixed has a failure-rate comparable to NASOQ-Range-Space and NASOQ-Tuned performs significantly better than NASOQ-Range-Space. Thus, using SoMod and the full-space method in NASOQ does not reduce the accuracy of the QP solver and can even improve accuracy with an appropriate choice of parameters for NASOQ-Tuned.

6.5.4 Effect of Numerical Range

NASOQ-Tuned and other QP solvers fail to solve 21 problem instances in our benchmark suite for the accuracy of 10^{-9}; Gurobi is an exception, but it can only solve

[6] The breakdown by application for each user-requested accuracy and for each solver is provided in Section A.4.

Table 6.2: Failure rate of NASOQ for different ranges of accuracy using range-space (NASOQ-Range-Space) and full-space (NASOQ-Fixed and NASOQ-Tuned) methods for small-scale problems in our QP repository. NASOQ-Fixed has a failure rate comparable to that of NASOQ-Range-Space. NASOQ-Tuned outperforms NASOQ-Range-Space and has no failures for accuracies $\epsilon = 10^{-3}$ and $\epsilon = 10^{-6}$.

	$\epsilon = 10^{-3}$	$\epsilon = 10^{-6}$	$\epsilon = 10^{-9}$
NASOQ-Range-Space	0%	0.23%	2.42%
NASOQ-Fixed	0.1511%	0.45%	1.44%
NASOQ-Tuned	0%	0%	0.91%

2 of these 21 problems. This section discusses properties of these 21 problems and explores why existing QP solvers and NASOQ-Tuned fail to solve them.

These problem instances have a large *numerical range* which can be classified into two categories: *(1)* problems that contain a large value (10^6 or larger) in either their input matrices or vectors (matrices H, A, and C and vectors q, b, and d in Equation 6.1); and *(2)* problems with large values in their primal or dual variables (vectors x, y, and z in Equations 6.1–6.2). This large numerical range limits the accuracy QP solvers can achieve in double precision.

This issue can be resolved if *(i)* for the first category, a scaling technique [54] is used to normalize the range, and *(ii)* for the second category, an implementation with higher precision is used; for example, using floating-point types with 128 bits of precision.

Gurobi is the only QP solver that converges for two of these problem instances. While NASOQ-Tuned is able to get close to the accuracy threshold of 10^{-9} (because the stationarity norm for these two problems in NASOQ-Tuned is 4.7×10^{-9} and 4.4×10^{-9}), the maximum value of the Lagrange multipliers in these two problems is about 10^6, which leads to inaccurate solutions for some intermediate KKT systems and thus leads to failure in NASOQ-Tuned.

6.5.5 Effect of SoMod

As discussed in Section 6.3, NASOQ uses SoMod to efficiently solve the successive KKT systems arising in active-set methods. In this section we analyze the effect of SoMod on NASOQ's performance and also separately demonstrate the efficiency of using LBL in NASOQ. We use the NASOQ-Fixed variant of NASOQ throughout this section because the effects of SoMod are the same in both variants. Figure 6.5 compares the performance profile of NASOQ-Fixed for $\epsilon = 10^{-9}$ with three different modifications of NASOQ: *(i)* NASOQ-Fixed-CHOLMOD, which uses CHOLMOD [26] instead of SoMod in NASOQ; *(ii)* NASOQ-Fixed-LBL, which instead of using row modification in NASOQ solves all KKT systems from scratch using LBL; and *(iii)* NASOQ-Fixed-

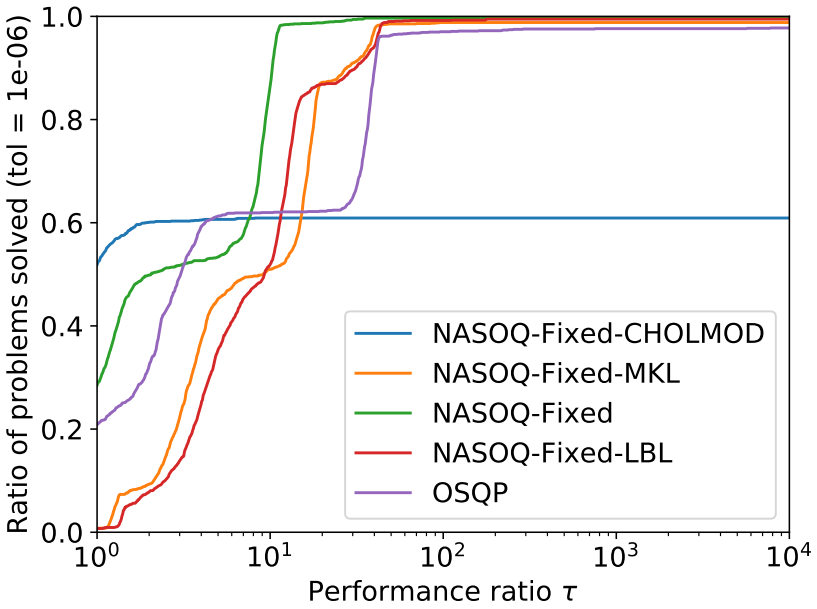


Figure 6.5: Performance profile of NASOQ using SoMod (NASOQ-Fixed), using CHOLMOD row modification (NASOQ-Fixed-CHOLMOD), solving from scratch using LBL (NASOQ-Fixed-LBL), and solving from scratch using MKL (NASOQ-Fixed-MKL). OSQP is also shown as a reference solver. NASOQ-Fixed (green line) performs better than the modified versions of NASOQ. Note that this performance profile contains both small and large QP instances, unlike Figure 6.4.

MKL which solves all KKT systems in NASOQ using the MKL-Pardiso solver. In all modifications, the same number of accuracy refinement iterations is used. Overall, NASOQ-Fixed is faster than all other implementations while achieving the fewest failures.

NASOQ-Fixed-CHOLMOD replaces SoMod's row modification and LBL phases with row modification and the solver used in CHOLMOD [26]; the iterative refinement from SoMod is still used in NASOQ-Fixed-CHOLMOD because CHOLMOD does not come with refinement. CHOLMOD's row modification primarily supports symmetric positive definite (SPD) matrices. CHOLMOD will fail or provide inaccurate results for some indefinite systems: *(i)* unlike SPD systems, the diagonal value of the L-factor in an indefinite KKT system can sometimes be negative, so CHOLMOD may fail for these systems as it uses square root in computations that involve the diagonal value; *(ii)* to update the new L-factor, CHOLMOD uses the already computed L-factors, and thus numerical errors and inaccuracies may propagate to subsequent computations. NASOQ however re-computes the affected columns using the input matrix; thus its L-factor is more accurate compared to that of CHOLMOD's as the number of iterations in the QP solver increase. Adding a perturbation value to the diagonal entries will remove some failures in CHOLMOD and so it can solve some (but not all) indefinite systems. With perturbation, the accuracy of the KKT solve using CHOLMOD is still

low and leads to failure in 40% of QP problems. As shown in Figure 6.5, NASOQ-Fixed-CHOLMOD mostly converges for small-scale QP problems when the number of variables is fewer than 50, and needs few iterations to converge. Unlike NASOQ, NASOQ-Fixed-CHOLMOD does not need to create an initial inclusive matrix and thus its initial setup time is small; this leads to faster performance for very small QP problem instances. NASOQ-Fixed is overall faster than NASOQ-Fixed-CHOLMOD and results in the fewest failures.

NASOQ-Fixed is on average 3.1× faster than NASOQ-Fixed-LBL and NASOQ-Fixed-MKL. This demonstrates the importance of using the factor modification method of SoMod in NASOQ to avoid solving the KKT systems from scratch. In addition, NASOQ-Fixed-LBL and NASOQ-Fixed-MKL demonstrate a similar performance profile which warrants the use of LBL as a replacement solver for MKL in SoMod while benefiting from the unique features of LBL that facilitate the implementation of row modification in SoMod.

To separately measure the performance of LBL in NASOQ, we use the indefinite solver from MKL-Pardiso instead of LBL for solving the initial KKT system in NASOQ; we call this variant NASOQ-Fixed-Initial-MKL. All other components of SoMod that solve the successive KKT systems remain unchanged. NASOQ-Fixed obtains a similar performance to that of NASOQ-Fixed-Initial-MKL: it is roughly 1.01× faster. The reason for the small effect of LBL on overall performance of NASOQ is that only a small fraction of the overall time is spent on the initial factorization; on average initial factorization only accounts for 25% of NASOQ time.

6.6 Related Work

QP algorithms can be categorized into three broad classes of methods: barrier (primarily interior-point), first-order, and active-set.

Barrier methods Barrier methods [64, 72, 122, 47, 134, 129, 135, 191] apply weighted barrier functions in the objective to enforce inequality constraints, converting inequality constrained QPs into equality-constrained nonlinear problems that can be solved by Newton or quasi-Newton methods [14, 23]. Performing a series (homotopy) of progressively tighter and thus more challenging barrier solves leads to interior-point and related methods [23]. As unconstrained optimization methods can then be directly applied, barrier methods can leverage sparse linear methods and so scale to large systems. Thus popular, a wide range of commercial [134, 129] and open-source [64, 190] interior-point solvers are available. However, accurate solutions are challeng-

ing to obtain for barrier methods – especially as system and constraint sizes grow. As accuracy is tightened, barrier solvers generally require increasingly large numbers of iterations. In turn, each iteration necessitates the expensive solution of a new, large-scale linear system.

First-order method: OSQP Barrier methods leverage second-order expansions of constraint information via Newton-type methods which can be expensive for computation per iterate. On the other hand, per-iteration efficient methods can be constructed by leveraging first-order strategies. In particular, operator splitting via the alternating direction method of multipliers (ADMM) has been recently applied to design OSQP [170], an efficient, highly scalable, first-order QP algorithm. OSQP forms all constraints into a single, large saddle-point-like system and then re-applies the solution of this system in each successive ADMM iteration to update primal and dual terms. OSQP thus can take advantage of lightweight computations per iteration and scales well to large, sparse QP problems. However, consistent with first-order strategies it can be slow or even unable to reach accurate solutions for larger and more challenging QP problems.

Active-set methods. Active-set methods [69, 159, 68, 120, 57] start with an initial feasible solution and then iterate to obtain the optimal solution while maintaining feasibility. After finding the initial feasible solution, which is either primal- or dual-feasible depending on the method, active-set methods look for the optimal active-set by solving successive KKT systems that include all constraints in the current active-set. Solving these KKT systems is the most expensive part in these methods. Active-set methods are divided into direct and indirect based on how they solve KKT systems [19]. Indirect methods, known as range-space [70] and null-space [68] methods, solve the KKT system using a Cholesky factorization along with a QR or Schur complement. Although these techniques provide an accurate solution, they do not preserve sparsity and thus do not scale for large QP problems due to high memory usage and extensive computations in the QR and Schur complement factorization. *Full-space methods* [75, 94], on the other hand, are direct active-set QP methods that work directly with the KKT system. Solving the KKT system using an iterative algorithm such as a Krylov subspace method [74] for active-set methods requires finding an efficient preconditioner for any arbitrary active-set which is often difficult [120]. Factorizing the KKT system using a direct method is very expensive and thus existing full-space techniques build an augmented system, along with an initial KKT, to compute the solution of the KKT system via the Schur complement [75] or Block-LU [94].

Both of these methods require large amounts of storage thus limiting their scalability and efficiency. Nevertheless, full-space methods have a promising property – they preserve the sparsity pattern of the system. In this work we leverage this sparsity to efficiently re-use factors across iterations. We introduce a new, full-space active-set algorithm, based on the Goldfarb-Idnani active-set strategy, that directly factorizes the successive KKT systems with SoMod and LBL to enable sparsity-oriented row modification and indefinite factorization of the successive KKT systems across our full-space QP solver's iterations.

LDL factorization Using direct factorization methods for solving a linear system of equations is common in many computer graphics applications [200, 87, 88] and is a subroutine in full-space QP solvers. A number of existing factorization methods are designed for solving a sparse SPD system of equations [26, 29, 28, 87]. These methods use a square-root based Cholesky factorization [37, 71] that will fail with symmetric indefinite systems from negative values under the square root when factorizing diagonals. Applying these solvers with regularization [87] prevents these types of failures but can introduce significant inaccuracies to problem solutions. Some existing indefinite factorization methods are square-root free [71] but are slow, e.g., SuiteSparse's LDL [32] which is a single-thread implementation. Parallel indefinite solvers such as MKL Pardiso [192], the standalone Pardiso solver [156, 154] and MA57 [52, 92] provide fast factorizations but do not support factor modifications for when a row/-column is changed. We introduce LBL, a new, parallel, indefinite, square-root free solver with pivoting, that additionally enables modifying already-computed factors efficiently. LBL extends the parallelism strategy from Cheshmi et al. [28] from SPD to indefinite factorizations where the now-required pivoting introduces new dependencies.

L-factor modification Modifying the L-factor to avoid re-factoring a symmetric matrix after small changes [39, 38] is a critical task in computer graphics [84, 87], circuit simulation [83, 40], and optimization [40]. In all such applications, the linear system of equations changes either by a rank update/downdate (adding or subtracting the outer product of a row by itself) or a row modification. Existing modification methods [39, 87] are generally designed to perform rank update/downdate ($A + ww^T$) on SPD matrices. These modification methods are then not applicable to our active-set QP solver where a row/column is modified for a symmetric indefinite system in each iteration. In turn, to our knowledge the only existing system with sparse row modification is CHOLMOD [40] which is an efficient solution designed for SPD matrices.

Thus CHOLMOD row modification does not provide accurate and stable solution updates for indefinite KKT systems; see Section 6.5.5. We propose SoMod row modification to leverage the sparsity pattern of constraint row updates and so accurately and efficiently modify the L-factor of indefinite factorization.

Chapter 7

Conclusion and Future Work

This thesis presents Sympiler, a domain-specific code generator that generates highly optimizes code for sparse matrix computations on modern parallel architectures. The thesis starts by introducing a novel symbolic decoupling strategy that enables the inspection of static sparsity patterns in sparse codes. It takes the sparse matrix pattern and the sparse matrix algorithm as inputs to perform symbolic analysis at compile time. It then uses the information from the symbolic analysis to apply a number of inspector-guided and low-level transformations to the sparse code. We demonstrate in Chapter 3 how this approach will lead to generating code that outperforms highly-optimized code from state-of-the-art libraries on single-core architecture on two sparse kernels.

Then in Chapter 4, we propose the Load-Balanced Level Coarsening algorithm inside ParSy that can be used as an inspector inside Sympiler to generate code that improves locality and reduces synchronization in sparse kernels when executing on parallel architectures. ParSy takes the numerical algorithm and sparsity pattern of the matrix and generates optimized parallel multi-core code. ParSy's inspector uses the LBC algorithm for inspection along with H-Level transformation for generating the code. ParSy-generated code outperforms state-of-the-art sparse libraries for sparse Cholesky and triangular solve across different multi-core processors.

The dissertation then addresses the problem of optimizing more than one sparse kernel at the same time. In chapter 5, we present sparse fusion and demonstrate how it improves parallelism, load balance, and data locality in sparse matrix combinations compared to when sparse kernels are optimized separately. Sparse fusion inspects the DAGs of the input sparse kernels and uses the MSP algorithm to balance the workload between wavefronts and determine whether to optimize data locality for within or between the kernels. Sparse fusion's generated code outperforms state-of-the-art implementations for sparse matrix optimizations.

The final chapter of the thesis aims to demonstrate that when sparsity patterns in certain applications change, the changes can be computed prior to optimizing the sparse matrix computation more efficiently. This is demonstrated with NASOQ and on QP solvers. NASOQ enables simultaneously accurate and efficient solves for large and sparse QP problems across application domains. To enable NASOQ we have constructed a new sparsity-oriented SoMod row modification method and LBL, our fast LDL factorization for indefinite systems. Together they enable the efficient updates and accurate solutions of the iteratively modified KKT systems critical to accurate QP solves. To better understand QP computational challenges and solver performance, we gathered a comprehensive benchmark set comprising a wide range of application-based QP problems. We hope that its application will lead to improved testing and further development of performant QP solvers.

As future work, the existing domain-specific compilation techniques, specifically what is proposed in this dissertation, can be used to improve the performance of general compilers. There are a relatively large number of domain-specific compilers that often create confusion amongst practitioners on when and how they can be used. Instead of presenting Sympiler to the community as yet another domain-specific code generator, I envision its approach to doing symbolic analysis to generate inspectors and code transformation for sparse codes can guide the design of future general-purpose compilers. Towards this direction, approaches that use program analysis to extract domain information for a general input code can be explored, and then this information can be used to automatically generate inspectors for symbolic analysis. I have started to investigate domain-specific and trace-based analysis to generate specialized code for sparse loops. Generative programming and just-in compilation can also be used to construct the general code specialization framework.

Optimizing matrix computation codes can lead to inaccuracy or incorrectness due to compile-time and/or run-time inefficiencies of the system. Verifying the optimization passes applied is either expensive due to the huge exploration space or impossible due to complex indexing in the codes. Even when verified, rounding errors in floating-point operations or transient faults specially when ran on large-scale and parallel machines can lead to inaccurate or wrong results. Techniques should be developed to ensure the correctness and resilience of the generated code while preserving its performance by proposing compile-time and runtime techniques that use domain-specific information or information from other abstractions to enable efficient verification at compile time. If such techniques are developed, they can be used to verify the transformations in Sympiler and facilitate their integration into general compilers.

Algorithms in several applications such as machine learning and scientific simulations are composed of several consecutive kernels and loops. Optimizing these loops jointly would enable opportunities to optimize for parallelism and locality. Fusing two loops in sparse iterative solvers is investigated in Sympiler however, there are potential opportunities in fusing more than two loops or computations. This extension would require investigating techniques that determine the profitability of loop fusion for more than two loops. For example, such analysis would decide which loops should be fused and in what order. This is an interesting future work direction as it extends the applicability of Sympiler to more applications.

Databases are critical in a wide class of applications ranging from multimedia retrievals such as Youtube and web content retrieval to scientific computing and global positioning systems. Efficiency and productivity are key to designing robust database applications. Achieving both requires a high-level language for productivity that at the same time provides the efficiency of low-level handwritten query plans. To obtain this, efficient query processing is necessary to convert the query plan corresponding to the high-level description into an efficient native code. Query processing has shifted from being I/O bound to becoming CPU bound because of hardware advancements that has led to larger main memories and non-volatile memory architecture, and due to the demand for computationally intensive analytics. Thus, database applications will significantly benefit from a query compiler that can apply workload-specific low-level optimization passes to generate an efficient code. For example, the compiler passes that are used for a spatial workload differ from those used for a graph processing workload. Generating codes that efficiently use all cores and vector units of each core is another requirement for a query compiler. Sympiler code specialization techniques can be be extended to support some of the database workloads.

Appendix A

Appendix for Transforming Sparse Matrix Computations

A.1 DAG Partitioners Limitations

Figure A.1 compares the performance of two DAG partitioners, DAGP and LBC for different sizes of sparse DAGs. In the one DAG configuration, the DAG paritioner partitions the DAG of sparse triangular solve (SpTRSV) CSR. In the joint DAG configuration, the DAG partitioner paritions the joint DAG of the sparse matrix vector multiplication (SpMV) CSR and SpTRSV CSR. Both configurations run on a set sparse matrices with different sizes from the SuiteSparse collection [41]. To compare the joint DAG configuration with the one DAG configurations, the x-axis shows the number of edges in one of the DAGs, i.e. SpTRSV DAG. The number of edges in the joint DAG is three times the edges of the SpTRSV DAG. As shown in the Figure A.1, the DAGP in both one DAG and joint DAG configurations are slower than LBC, for both small and large size DAGs. Also, DAGP on the joint DAG runs out of memory for last seven large DAGs (hence not shown in the figure).

Figure A.2 separately shows partitioning time of LBC. As shown, the execution time of LBC is not proportional to the number of edges and the execution time is significantly higher for most of joint DAGs and for some large DAGs. For example, while the number of edges in the joint DAG is three times that of one DAG, LBC on the joint DAG is slower than LBC on one DAG with an average of 9.2×. Chordalizion phase of initial coarsening in LBC is the reason behind this non-linear slowdown. Because LBC's partitioning time is reasonably low for one DAG, sparse fusion uses it to partition one of the DAGs, i.e. the head DAG.

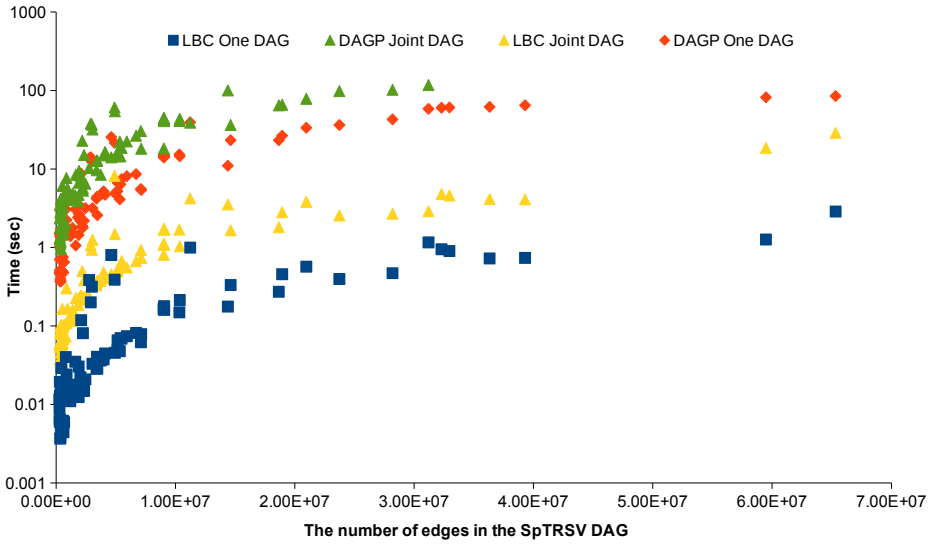


Figure A.1: Performance of DAGP and LBC DAG partitioners for DAGs with different number of edges in an individual and joint DAG.

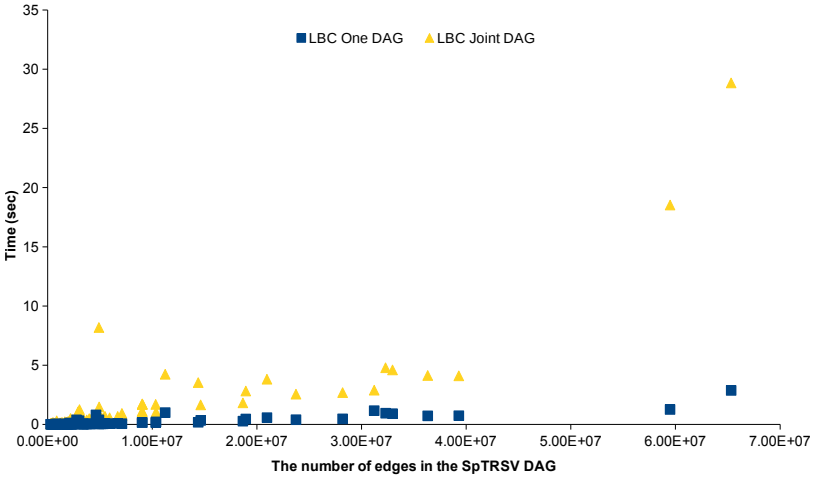


Figure A.2: Performance of LBC DAG partitioner for one DAG and joint DAG.

A.2 Experimental Results for Xeon Platinum8160

Figure A.3 shows the performance of the fused code from sparse fusion, the unfused implementation from ParSy and MKL, and the fused wavefront, fused LBC, and fused DAGp implementations. All execution times are normalized over a *baseline*. The baseline is obtained by running each kernel individually with a sequential implementation. As shown, sparse fusion on the Platinum processor outperforms other implementations with a similar trend to results on Xeon E5 processors, which are discussed in Section 4 of Chapter 5. The sparse fusion's fused code is on average 1.3× faster than ParSy's executor code and 5.2× faster than MKL across all kernel combinations (excluding ILU0 combinations). The fused code from sparse fusion is

on average 1.9×, 5.2×, and 7.4× faster than in order fused wavefront, fused LBC, and fused DAGp.

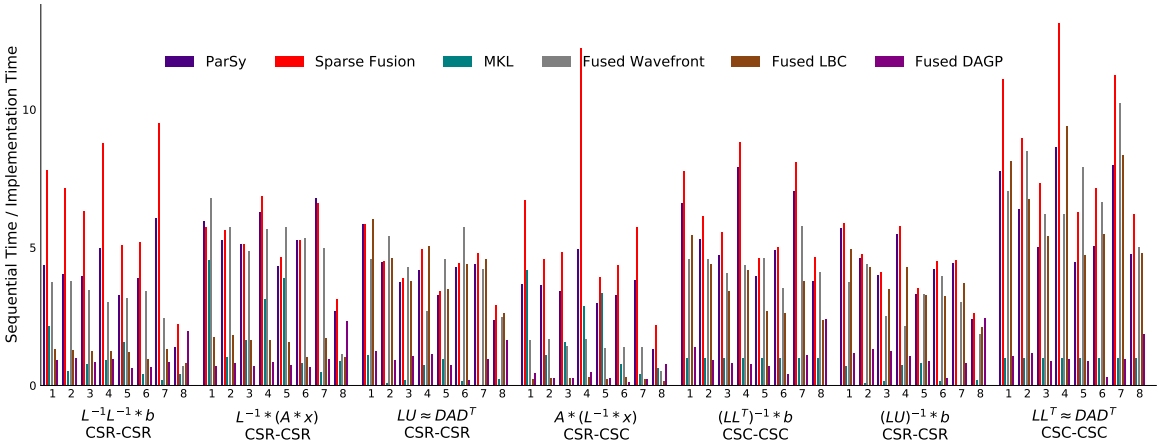


Figure A.3: Performance of different implementations shown with speedup from dividing baseline time by implementation time. The target architecture is Xeon Platinum8160 with 24 cores.

Figure A.4 shows the number of times that the executor should run to amortize the cost of inspection for implementations that have an inspector. As shown, similar to Figure 5.9, sparse fusion has the lowest overhead compared to other methods and DAGP and LBC on a joint DAG have the most expensive inspection overhead.

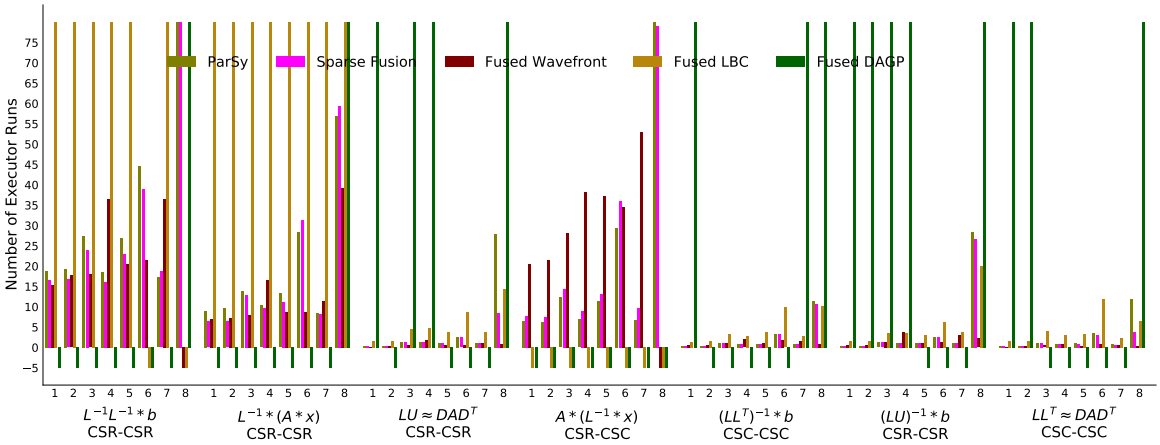


Figure A.4: The number of executor runs to amortize inspector cost. Values are clipped between -5 and 80. (lower is better)

A.3 Settings for QP solvers

In this section we explain settings used for each solver and also discuss how these settings provide a fair evaluation. Throughout this section, `eps` refers to the accuracy threshold ϵ which is either 10^{-3}, 10^{-6}, or 10^{-9}.

A.3.1 Gurobi

The following parameters are set to `eps` for Gurobi [1]:

```
model_.set(GRB_DoubleParam_OptimalityTol, eps);
model_.set(GRB_DoubleParam_FeasibilityTol, eps);
```

Gurobi supports using an absolute termination criteria and setting these parameters provides a fair comparison with other solvers.

A.3.2 MOSEK

The following parameters are set to `eps` in MOSEK.[2]

```
program->setParamDouble(MSKdparam_enum::MSK_DPAR_INTPNT_QO_TOL_DFEAS,eps);
program->setParamDouble(MSKdparam_enum::MSK_DPAR_INTPNT_QO_TOL_PFEAS,eps);
program->setParamDouble(MSKdparam_enum::MSK_DPAR_INTPNT_QO_TOL_MU_RED,eps);
program->setParamDouble(MSKdparam_enum::MSK_DPAR_INTPNT_QO_TOL_INFEAS,eps);
```

MOSEK does not support an absolute termination criteria thus in addition to the demonstrated configuration, i.e., using `eps` for all parameters, we tested MOSEK with three different settings to find the best configuration for a fair enough evaluation:

- setting all `eps` to 10^{-16}.
- setting `MSKdparam_enum::MSK_DPAR_INTPNT_QO_TOL_INFEAS` to 10^{-12} and the remaining three parameters to `eps`.
- setting all parameters to their default values for all requested accuracy thresholds, i.e., `eps`.

Our experiments show that using `eps` for all parameters provides an overall better failure rate and performance.

In addition to the above, we also realized using `eps` for parameter `MSKdparam_enum::MSK_DPAR_INTPNT_QO_TOL_REL_GAP` increases MOSEK's failure rate, so we did not configure this parameter and used its default value instead.

Parameter `MSKdparam_enum::MSK_DPAR_INTPNT_QO_TOL_NEAR_REL` shows when the MOSEK computed solution is optimal. This does not affect the number of iterations in MOSEK. We set it to `1.0` and use unified routines, which is used across solvers, to measure optimality for fair comparison.

[1] The specification of MOSEK termination criteria can be obtained from: https://www.gurobi.com/documentation/8.1/refman/parameters.html

[2] The specification of MOSEK's termination criteria can be obtained from: https://docs.mosek.com/9.0/toolbox/param-groups.html#doc-termination-param-pargroup

A.3.3 OSQP

OSQP setting used in our experiments are: [3]

```
settings->linsys_solver = MKL_PARDISO_SOLVER;
settings->eps_abs=eps;
settings->eps_rel=eps;
settings->max_iter=40000000;
settings->eps_prim_inf = eps;
settings->eps_dual_inf = eps;
settings->polish = is_polish; // is set to 1 for OSQP-polished.
settings->verbose = 0;
settings->time_limit = 2000.0;
```

OSQP's default termination criteria is a relative measure, thus, we modify the OSQP code to support an absolute termination criteria for fair evaluation. Line 725 and 737 of `src/auxil.c` are changed and instead of calling `compute_pri_tol(work, eps_abs, eps_rel)` and `compute_dua_tol(work, eps_abs, eps_rel)` an absolute threshold is used with `eps_prim = eps_abs` and `eps_dual = eps_abs`. Functions `compute_pri_tol(work, eps_abs, eps_rel)` and `compute_dua_tol(work, eps_abs, eps_rel)` compute a relative threshold for primal and dual variables respectively. The modified code is provided with this document.

A.3.4 QL

For QL, we set the input `eps` parameter to the accuracy threshold `eps`.

A.4 Application-based breakdown for NASOQ

In this section, we show the failure rate and speedup of NASOQ compared to other solvers. To show the speedup, we use a geometric mean (GM). Speedup numbers are normalized to the geometric mean of NASOQ-Tuned, a larger the speedup value corresponds to a slower solver compared to NASOQ-Tuned.

Tables 1-4 show the results for four classes of QP problems in our QP repository. The total number of QP problems is 1513 which include 1308 contact simulation (Table A.2), 53 Maros-Mészáros (Table A.3), 120 model predictive control(Table A.4), and 32 object deformation and model reconstruction (Table A.5) problems.

[3] The osqp settings are available via this link, https://osqp.org/docs/interfaces/solver_settings.html.

	$\epsilon = 10^{-3}$			$\epsilon = 10^{-6}$			$\epsilon = 10^{-9}$		
	Failue (%)	rate	Speedup (GM)	Failue (%)	rate	Speedup (GM)	Failue (%)	rate	Speedup (GM)
Gurobi	1.1		3.4	3.9		6.8	14.3		27.5
MOSEK	13.5		60.7	92.3		>3000	97.3		>3000
NASOQ-Fixed	0.4		1.2	1		1.7	2.3		1.4
NASOQ-Tuned	0		1	0		1	1.5		1
OSQP	2.2		3.3	3		3.9	14.4		26.3
OSQP-polished	0.9		1.7	2		2.8	14		24.8

Table A.1: All problems.

	$\epsilon = 10^{-3}$			$\epsilon = 10^{-6}$			$\epsilon = 10^{-9}$		
	Failue (%)	rate	Speedup (GM)	Failue (%)	rate	Speedup (GM)	Failue (%)	rate	Speedup (GM)
Gurobi	0.6		3.5	3.5		8.2	10.1		21.8
MOSEK	13.5		87.2	94.3		>3000	98.1		>3000
NASOQ-Fixed	0		1	0.2		1.2	0.8		1.1
NASOQ-Tuned	0		1	0		1	0.6		1
OSQP	1.2		2.5	1.1		2.1	13.8		42.6
OSQP-polished	0.6		1.8	0.6		1.7	13.8		42.9

Table A.2: Contact simulation problems.

	$\epsilon = 10^{-3}$			$\epsilon = 10^{-6}$			$\epsilon = 10^{-9}$		
	Failue (%)	rate	Speedup (GM)	Failue (%)	rate	Speedup (GM)	Failue (%)	rate	Speedup (GM)
Gurobi	13.2		9.8	15.1		8.9	58.5		2166.7
MOSEK	34		136.8	79.2		>3000	88.7		>3000
NASOQ-Fixed	9.4		3.2	24.5		28.8	35.8		25.3
NASOQ-Tuned	0		1	0		1	15.1		1
OSQP	24.5		31.9	45.3		989	47.1		182.7
OSQP-polished	9.4		1.9	35.8		183	40		45.2

Table A.3: Maros-Mészáros problems.

	$\epsilon = 10^{-3}$			$\epsilon = 10^{-6}$			$\epsilon = 10^{-9}$		
	Failue (%)	rate	Speedup (GM)	Failue (%)	rate	Speedup (GM)	Failue (%)	rate	Speedup (GM)
Gurobi	0		0.4	0.8		34.3	19.2		>3000
MOSEK	0.8		0.4	86.7		>3000	100		>3000
NASOQ-Fixed	0		1	0		1	0		1
NASOQ-Tuned	0		1	0		1	0		1
OSQP	3.3		12.7	2.5		119.3	2.5		123
OSQP-polished	0		61.1	0		3.4	0		4

Table A.4: Model Predictive Control (MPC) problems.

	$\epsilon = 10^{-3}$		$\epsilon = 10^{-6}$		$\epsilon = 10^{-9}$	
	Failue rate (%)	Speedup (GM)	Failue rate (%)	Speedup (GM)	Failue rate (%)	Speedup (GM)
Gurobi	6.3	10.4	12.5	24	90.7	>3000
MOSEK	28.1	231.3	50	>3000	68.8	>3000
NASOQ-Fixed	3.1	1.7	0	1	18.8	1
NASOQ-Tuned	0	1	0	1	18.8	1
OSQP	3.1	4.2	12.5	22.8	28.1	6.2
OSQP-polished	0	2.2	12.5	23.3	34.4	19.6

Table A.5: Model reconstruction and object deformation problems.

Bibliography

[1] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J. Zico Kolter. "Differentiable Convex Optimization Layers". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019.

[2] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. "Are static schedules so bad? A case study on Cholesky factorization". In: *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE. 2016, pp. 1021–1030.

[3] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects". In: *Journal of Physics: Conference Series*. Vol. 180. 1. IOP Publishing. 2009, p. 012037.

[4] José I. Aliaga, Joaquín Pérez, and Enrique S. Quintana-Ortí. "Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers". In: *Euro-Par 2015: Parallel Processing*. Ed. by Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 675–686. ISBN: 978-3-662-48096-0.

[5] Martin S Alnæs, Anders Logg, Kristian B Ølgaard, Marie E Rognes, and Garth N Wells. "Unified form language: A domain-specific language for weak formulations of partial differential equations". In: *ACM Transactions on Mathematical Software (TOMS)* 40.2 (2014), p. 9.

[6] Patrick R Amestoy, Iain S Duff, and J-Y L'Excellent. "Multifrontal parallel distributed symmetric and unsymmetric solvers". In: *Computer methods in applied mechanics and engineering* 184.2 (2000), pp. 501–520.

[7] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. "A fully asynchronous multifrontal solver using distributed dynamic scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41.

[8] Patrick R Amestoy, Abdou Guermouche, Jean-Yves L'Excellent, and Stéphane Pralet. "Hybrid scheduling for the parallel solution of linear systems". In: *Parallel computing* 32.2 (2006), pp. 136–156.

[9] Brandon Amos and J. Zico Kolter. "OptNet: Differentiable Optimization as a Layer in Neural Networks". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017.

[10] Corinne Ancourt and François Irigoin. "Scanning polyhedra with DO loops". In: *ACM Sigplan Notices*. Vol. 26. 7. ACM. 1991, pp. 39–50.

[11] M. Arioli, I. S. Duff, S. Gratton, and S. Pralet. "A Note on GMRES Preconditioned by a Perturbed LDL^T Decomposition with Static Pivoting". In: *SIAM J. Sci. Comput.* 29.5 (Sept. 2007). ISSN: 1064-8275.

[12] Cleve Ashcraft and Roger G Grimes. "SPOOLES: An Object-Oriented Sparse Matrix Library." In: *PPSC*. 1999.

[13] Yunfei Bai, M. Danny Kaufman, C. Karen Liu, and Jovan Popović. "Artistic-dynamics for 2D animation". In: *ACM Transactions on Graphics (SIGGRAPH 2016)*. 2016.

[14] AS El-Bakry, Richard A Tapia, T Tsuchiya, and Yin Zhang. "On the formulation and theory of the Newton interior-point method for nonlinear programming". In: *Journal of Optimization Theory and Applications* 89.3 (1996).

[15] Jernej Barbic. "Real-Time Reduced Large-Deformation Models and Distributed Contact for Computer Graphics and Haptics". PhD thesis. USA, 2007. ISBN: 9780549203131.

[16] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors". In: *PPOPP* 44.4 (2009), pp. 219–228.

[17] Christopher Batty, Florence Bertails, and Robert Bridson. "A Fast Variational Framework for Accurate Solid-Fluid Coupling". In: *ACM Trans. Graph.* 26.3 (July 2007), 100–es. ISSN: 0730-0301. DOI: 10.1145/1276377.1276502. URL: https://doi.org/10.1145/1276377.1276502.

[18] Michele Benzi, Jane K Cullum, and Miroslav Tuma. "Robust approximate inverse preconditioning for the conjugate gradient method". In: *SIAM Journal on Scientific Computing* 22.4 (2000), pp. 1318–1332.

[19] Michele Benzi, Gene H Golub, and Jörg Liesen. "Numerical solution of saddle point problems". In: *Acta numerica* 14 (2005).

[20] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. "Ebb: A DSL for Physical Simulation on CPUs and GPUs". In: *ACM Trans. Graph.* 35.2 (May 2016), 21:1–21:12. ISSN: 0730-0301. DOI: 10.1145/2892632. URL: http://doi.acm.org/10.1145/2892632.

[21] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. "Pluto: A practical and fully automatic polyhedral program optimization system". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer. 2008.

[22] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends in Machine Learning* 3.1 (2011).

[23] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[24] Erin Claire Carson. "Communication-avoiding Krylov subspace methods in theory and practice". PhD thesis. UC Berkeley, 2015.

[25] Chun Chen. "Polyhedra scanning revisited". In: *ACM SIGPLAN Notices* 47.6 (2012), pp. 499–508.

[26] Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. "Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate". In: *ACM Transactions on Mathematical Software (TOMS)* 35.3 (2008), p. 22.

[27] Kazem Cheshmi, Leila Cheshmi, and Maryam Mehri Dehnavi. "Sparsity-Aware Storage Format Selection". In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. 2018, pp. 1034–1037. DOI: 10.1109/HPCS.2018.00162.

[28] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. "ParSy: Inspection and Transformation of Sparse Matrix Computations for Parallelism". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. Dallas, Texas: IEEE Press, 2018.

[29] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. "Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: ACM, 2017. ISBN: 978-1-4503-5114-0.

[30] Kazem Cheshmi, Danny M Kaufman, Shoaib Kamil, and Maryam Mehri Dehnavi. "NASOQ: numerically accurate sparsity-oriented QP solver". In: *ACM Transactions on Graphics (TOG)* 39.4 (2020), pp. 96–1.

[31] Edward G Coffman Jr, Michael R Garey, and David S Johnson. "An application of bin-packing to multiprocessor scheduling". In: *SIAM Journal on Computing* 7.1 (1978), pp. 1–17.

[32] Timothy A Davis. "Algorithm 1000: SuiteSparse: GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra". In: *ACM Transactions on Mathematical Software (TOMS)* 45.4 (2019).

[33] Timothy A Davis. "Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method". In: *ACM Transactions on Mathematical Software (TOMS)* 30.2 (2004), pp. 196–199.

[34] Timothy A Davis. "Algorithm 849: A concise sparse Cholesky factorization package". In: *ACM Transactions on Mathematical Software (TOMS)* 31.4 (2005), pp. 587–591.

[35] Timothy A Davis. "Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), p. 8.

[36] Timothy A Davis. "Algorithm 930: FACTORIZE: An object-oriented linear system solver for MATLAB". In: *ACM Transactions on Mathematical Software (TOMS)* 39.4 (2013), p. 28.

[37] Timothy A Davis. *Direct methods for sparse linear systems*. Vol. 2. Siam, 2006.

[38] Timothy A Davis and William W Hager. "Dynamic supernodes in sparse Cholesky update/-downdate and triangular solves". In: *ACM Transactions on Mathematical Software (TOMS)* 35.4 (2009), p. 27.

[39] Timothy A Davis and William W Hager. "Modifying a sparse Cholesky factorization". In: *SIAM Journal on Matrix Analysis and Applications* 20.3 (1999).

[40] Timothy A Davis and William W Hager. "Row modifications of a sparse Cholesky factorization". In: *SIAM Journal on Matrix Analysis and Applications* 26.3 (2005), pp. 621–639.

[41] Timothy A Davis and Yifan Hu. "The University of Florida sparse matrix collection". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), p. 1.

[42] Timothy A. Davis and Ekanathan Palamadai Natarajan. "Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems". In: *ACM Trans. Math. Softw.* 37.3 (Sept. 2010), 36:1–36:17. ISSN: 0098-3500. DOI: 10.1145/1824801.1824814. URL: http://doi.acm.org/10.1145/1824801.1824814.

[43] Maryam Mehri Dehnavi, David M Fernández, and Dennis Giannacopoulos. "Enhancing the performance of conjugate gradient solvers on graphic processing units". In: *IEEE Transactions on Magnetics* 47.5 (2011), pp. 1162–1165.

[44] James W Demmel, Stanley C Eisenstat, John R Gilbert, Xiaoye S Li, and Joseph WH Liu. "A supernodal approach to sparse partial pivoting". In: *SIAM Journal on Matrix Analysis and Applications* 20.3 (1999), pp. 720–755.

[45] James W Demmel, John R Gilbert, and Xiaoye S Li. "An asynchronous parallel supernodal algorithm for sparse gaussian elimination". In: *SIAM Journal on Matrix Analysis and Applications* 20.4 (1999), pp. 915–952.

[46] Elizabeth D Dolan and Jorge J Moré. "Benchmarking optimization software with performance profiles". In: *Mathematical programming* 91.2 (2002).

[47] A. Domahidi, E. Chu, and S. Boyd. "ECOS: An SOCP solver for embedded systems". In: *2013 European Control Conference (ECC)*. 2013.

[48] Richard C Dorf. *Electronics, power electronics, optoelectronics, microwaves, electromagnetics, and radar*. CRC press, 2006.

[49] Iain S Duff, Nick IM Gould, John K Reid, Jennifer A Scott, and Kathryn Turner. "The factorization of sparse symmetric indefinite matrices". In: *IMA Journal of Numerical Analysis* 11.2 (1991), pp. 181–204.

[50] Iain S Duff and John K Reid. "The multifrontal solution of indefinite sparse symmetric linear". In: *ACM Transactions on Mathematical Software (TOMS)* 9.3 (1983), pp. 302–325.

[51] Iain S Duff and John Ker Reid. "The design of MA48: a code for the direct solution of sparse unsymmetric linear systems of equations". In: *ACM Transactions on Mathematical Software (TOMS)* 22.2 (1996), pp. 187–226.

[52] Iain S. Duff. "MA57—a Code for the Solution of Sparse Symmetric Definite and Indefinite Systems". In: *ACM Trans. Math. Softw.* 30.2 (June 2004). ISSN: 0098-3500.

[53] Marek Dvorožňák, Saman Sepehri Nejad, Ondřej Jamriška, Alec Jacobson, Ladislav Kavan, and Daniel Sýkora. "Seamless Reconstruction of Part-Based High-Relief Models from Hand-Drawn Images". In: *Proceedings of the Joint Symposium on Computational Aesthetics and Sketch-Based Interfaces and Modeling and Non-Photorealistic Animation and Rendering*. Expressive '18. Victoria, British Columbia, Canada: Association for Computing Machinery, 2018. ISBN: 9781450358927.

[54] Joseph M Elble and Nikolaos V Sahinidis. "Scaling linear optimization problems prior to application of the simplex method". In: *Computational Optimization and Applications* 52.2 (2012), pp. 345–371.

[55] Perry A Emrath, S Chosh, and David A Padua. "Event synchronization analysis for debugging parallel programs". In: *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. ACM. 1989, pp. 580–588.

[56] Kenny Erleben. "Numerical methods for linear complementarity problems in physics-based animation". In: *Acm Siggraph 2013 Courses*. 2013.

[57] Hans Joachim Ferreau, Christian Kirches, Andreas Potschka, Hans Georg Bock, and Moritz Diehl. "qpOASES: A parametric active-set algorithm for quadratic programming". In: *Mathematical Programming Computation* 6.4 (2014).

[58] M Fesanghary, Mehrdad Mahdavi, M Minary-Jolandan, and Y Alizadeh. "Hybridizing harmony search algorithm with sequential quadratic programming for engineering optimization problems". In: *Computer methods in applied mechanics and engineering* 197.33-40 (2008).

[59] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[60] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[61] Alan George and Joseph W Liu. "Computer solution of large sparse positive definite". In: (1981).

[62] Alan George and Joseph WH Liu. "The design of a user interface for a sparse matrix package". In: *ACM Transactions on Mathematical Software (TOMS)* 5.2 (1979), pp. 139–162.

[63] Alan George, Joseph WH Liu, and Esmond Ng. "Communication results for parallel sparse Cholesky factorization on a hypercube". In: *Parallel Computing* 10.3 (1989), pp. 287–298.

[64] E. Michael Gertz and Stephen J. Wright. "Object-Oriented Software for Quadratic Programming". In: *ACM Trans. Math. Softw.* 29.1 (Mar. 2003). ISSN: 0098-3500.

[65] Pieter Ghysels and Wim Vanroose. "Hiding global synchronization latency in the preconditioned conjugate gradient algorithm". In: *Parallel Computing* 40.7 (2014), pp. 224–238.

[66] John R Gilbert and Tim Peierls. "Sparse partial pivoting in time proportional to arithmetic operations". In: *SIAM Journal on Scientific and Statistical Computing* 9.5 (1988), pp. 862–874.

[67] John R Gilbert and Robert Schreiber. "Highly parallel sparse Cholesky factorization". In: *SIAM Journal on Scientific and Statistical Computing* 13.5 (1992), pp. 1151–1172.

[68] Philip E Gill, Walter Murray, Michael A Saunders, and Elizabeth Wong. *User guide for SQOPT 7: Software for large-scale linear and quadratic programming*. 2005.

[69] Philip E Gill, Walter Murray, Michael A Saunders, and Margaret H Wright. "Inertia-controlling methods for general quadratic programming". In: *Siam Review* 33.1 (1991).

[70] Donald Goldfarb and Ashok Idnani. "A numerically stable dual method for solving strictly convex quadratic programs". In: *Mathematical programming* 27.1 (1983).

[71] Gene H Golub and Charles F Van Loan. *Matrix computations*. Vol. 3. JHU press, 2012.

[72] Jacek Gondzio and Andreas Grothey. "Solving nonlinear financial planning problems with 109 decision variables on massively parallel architectures". In: *WIT Transactions on Modelling and Simulation* 43 (2006).

[73] Nicholas Gould. *An introduction to algorithms for continuous optimization*. 2006.

[74] Nicholas IM Gould, Mary E Hribar, and Jorge Nocedal. "On the solution of equality constrained quadratic programming problems arising in optimization". In: *SIAM Journal on Scientific Computing* 23.4 (2001).

[75] Nicholas IM Gould, Dominique Orban, and Philippe L Toint. "GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization". In: *ACM Transactions on Mathematical Software (TOMS)* 29.4 (2003).

[76] Nicholas IM Gould, Jennifer A Scott, and Yifan Hu. "A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations". In: *ACM Transactions on Mathematical Software (TOMS)* 33.2 (2007), p. 10.

[77] R Govindarajan and Jayvant Anantpur. "Runtime dependence computation and execution of loops on heterogeneous systems". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society. 2013, pp. 1–10.

[78] Laura Grigori and Sophie Moufawad. "Communication avoiding ILU0 preconditioner". In: *SIAM Journal on Scientific Computing* 37.2 (2015), pp. C217–C246.

[79] Gaël Guennebaud and Benoit Jacob. "Eigen". In: *URl: http://eigen. tuxfamily. org* (2010).

[80] John A Gunnels, Fred G Gustavson, Greg M Henry, and Robert A Van De Geijn. "FLAME: Formal linear algebra methods environment". In: *ACM Transactions on Mathematical Software (TOMS)* 27.4 (2001), pp. 422–455.

[81] Anshul Gupta, George Karypis, and Vipin Kumar. "Highly scalable parallel algorithms for sparse matrix factorization". In: *IEEE Transactions on Parallel and Distributed Systems* 8.5 (1997), pp. 502–520.

[82] Fred G Gustavson, Werner Liniger, and R Willoughby. "Symbolic generation of an optimal Crout algorithm for sparse systems of linear equations". In: *Journal of the ACM (JACM)* 17.1 (1970), pp. 87–109.

[83] William W Hager. "Updating the inverse of a matrix". In: *SIAM review* 31.2 (1989).

[84] Florian Hecht, Yeon Jin Lee, Jonathan R. Shewchuk, and James F. O'Brien. "Updated Sparse Cholesky Factors for Corotational Elastodynamics". In: *ACM Trans. Graph.* 31.5 (Sept. 2012). ISSN: 0730-0301.

[85] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2017.

[86] Pascal Hénon, Pierre Ramet, and Jean Roman. "PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems". In: *Parallel Computing* 28.2 (2002), pp. 301–321.

[87] Philipp Herholz and Marc Alexa. "Factor Once: Reusing Cholesky Factorizations on Sub-Meshes". In: *ACM Trans. Graph.* 37.6 (Dec. 2018). ISSN: 0730-0301.

[88] Philipp Herholz, Timothy A. Davis, and Marc Alexa. "Localized Solutions of Sparse Linear Systems for Geometry Processing". In: *ACM Trans. Graph.* 36.6 (Nov. 2017). ISSN: 0730-0301.

[89] Julien Herrmann, M Yusuf Ozkaya, Bora Uçar, Kamer Kaya, and Ümit VV Çatalyürek. "Multilevel algorithms for acyclic partitioning of directed acyclic graphs". In: *SIAM Journal on Scientific Computing* 41.4 (2019), A2117–A2145.

[90] Toby Heyn, Mihai Anitescu, Alessandro Tasora, and Dan Negrut. "Using Krylov subspace and spectral methods for solving complementarity problems in many-body contact dynamics simulation". In: *International Journal for Numerical Methods in Engineering* 95.7 (2013).

[91] Mark Frederick Hoemmen et al. "Communication-avoiding Krylov subspace methods". In: (2010).

[92] Jonathan D. Hogg and Jennifer A. Scott. "Pivoting Strategies for Tough Sparse Indefinite Systems". In: *ACM Trans. Math. Softw.* 40.1 (Oct. 2013). ISSN: 0098-3500.

[93] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. "High-performance Code Generation for Stencil Computations on GPU Architectures". In: *Proceedings of the 26th ACM International Conference on Supercomputing*. ICS '12. San Servolo Island, Venice, Italy: ACM, 2012, pp. 311–320. ISBN: 978-1-4503-1316-2. DOI: 10.1145/2304576.2304619.

[94] Hanh M Huynh. *A large-scale quadratic programming solver based on block-LU updates of the KKT system*. Tech. rep. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 2008.

[95] Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. "Bounded Biharmonic Weights for Real-Time Deformation". In: *ACM Trans. Graph.* SIGGRAPH '11 (July 2011).

[96] Alec Jacobson, Daniele Panozzo, et al. *libigl: A simple C++ geometry processing library*. https://libigl.github.io/. 2018.

[97] Carlo Janna, Massimiliano Ferronato, and Giuseppe Gambolati. "The use of supernodes in factored sparse approximate inverse preconditioning". In: *SIAM Journal on Scientific Computing* 37.1 (2015), pp. C72–C94.

[98] David S Johnson. "Fast algorithms for bin packing". In: *Journal of Computer and System Sciences* 8.3 (1974), pp. 272–314.

[99] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[100] Sam Kamin, María Jesús Garzarán, Barış Aktemur, Danqing Xu, Buse Yılmaz, and Zhongbo Chen. "Optimization by Runtime Specialization for Sparse Matrix-Vector Multiplication". In: *SIGPLAN Not.* 50.3 (Sept. 2014), pp. 93–102. ISSN: 0362-1340. DOI: 10.1145/2775053.2658773. URL: https://doi.org/10.1145/2775053.2658773.

[101] G. Karypis. "METIS : Unstructured graph partitioning and sparse matrix ordering system". In: *Technical Report* (1997).

[102] George Karypis and Vipin Kumar. "A high performance sparse Cholesky factorization algorithm for scalable parallel computers". In: *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers' 95., Fifth Symposium on the*. IEEE. 1995, pp. 140–147.

[103] George Karypis and Vipin Kumar. "A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices". In: *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN* (1998).

[104] Danny M. Kaufman, Shinjiro Sueda, Doug L. James, and Dinesh K. Pai. "Staggered Projections for Frictional Contact in Multibody Systems". In: *ACM Trans. Graph.* SIGGRAPH Asia '08 (Dec. 2008).

[105] Wayne Kelly. "Optimization within a unified transformation framework". In: (1998).

[106] David S Kershaw. "The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations". In: *Journal of computational physics* 26.1 (1978), pp. 43–65.

[107] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. "Simit: A language for physical simulation". In: *ACM Transactions on Graphics (TOG)* 35.2 (2016), p. 20.

[108] Christopher D Krieger, Michelle Mills Strout, Catherine Olschanowsky, Andrew Stone, Stephen Guzik, Xinfeng Gao, Carlo Bertolli, Paul HJ Kelly, Gihan Mudalige, Brian Van Straalen, et al. "Loop chaining: A programming abstraction for balancing locality and parallelism". In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE. 2013, pp. 375–384.

[109] David I.W. Levin. *GAUSS Library*. https://github.com/dilevin/GAUSS. 2019.

[110] Ruipeng Li and Yousef Saad. "GPU-accelerated preconditioned iterative linear solvers". In: *The Journal of Supercomputing* 63.2 (2013), pp. 443–466.

[111] Xiaoye S Li. "An overview of SuperLU: Algorithms, implementation, and user interface". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 302–325.

[112] Xiaoye S Li and James W Demmel. "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems". In: *ACM Transactions on Mathematical Software (TOMS)* 29.2 (2003), pp. 110–140.

[113] Douglas K Lilly. "On the computational stability of numerical solutions of time-dependent non-linear geophysical fluid dynamics problems". In: *Mon. Wea. Rev* 93.1 (1965), pp. 11–26.

[114] Amy W Lim, Gerald I Cheong, and Monica S Lam. "An affine partitioning algorithm to maximize parallelism and minimize communication". In: *Proceedings of the 13th international conference on Supercomputing*. ACM. 1999, pp. 228–237.

[115] Bangtian Liu, Kazem Cheshmi, Saeed Soori, Michelle Mills Strout, and Maryam Mehri Dehnavi. "MatRox: Modular Approach for Improving Data Locality in Hierarchical (Mat)Rix App(Rox)Imation". In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '20. San Diego, California: Association for Computing Machinery, 2020, pp. 389–402. ISBN: 9781450368186. DOI: 10.1145/3332466.3374548. URL: https://doi.org/10.1145/3332466.3374548.

[116] Joseph W. H. Liu. "The Role of Elimination Trees in Sparse Factorization". In: *SIAM J. Matrix Anal. Appl.* 11.1 (Jan. 1990), pp. 134–172. ISSN: 0895-4798. DOI: 10.1137/0611010. URL: http://dx.doi.org/10.1137/0611010.

[117] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. "A synchronization-free algorithm for parallel sparse triangular solves". In: *European Conference on Parallel Processing*. Springer. 2016, pp. 617–630.

[118] José A Zevallos Luna, Alexandre Siligaris, Cédric Pujol, and Laurent Dussopt. "A packaged 60 GHz low-power transceiver with integrated antennas for short-range communications." In: *Radio and Wireless Symposium*. 2013, pp. 355–357.

[119] Fabio Luporini, David A Ham, and Paul HJ Kelly. "An algorithm for the optimization of finite element integration loops". In: *arXiv preprint arXiv:1604.05872* (2016).

[120] Christopher Mario Maes. "A regularized active-set method for sparse convex quadratic programming". PhD thesis. USA: Stanford University, 2011.

[121] Istvan Maros and Csaba Mészáros. "A repository of convex quadratic programming problems". In: *Optimization Methods and Software* 11.1-4 (1999).

[122] Jacob Mattingley and Stephen Boyd. "CVXGEN: A code generator for embedded convex optimization". In: *Optimization and Engineering* 13.1 (2012).

[123] M. MehriDehnavi, Y. El-Kurdi, J. Demmel, and D. Giannacopoulos. "Communication-Avoiding Krylov Techniques on Graphic Processing Units". In: *IEEE Transactions on Magnetics* 49.5 (2013), pp. 1749–1752. DOI: 10.1109/TMAG.2013.2244861.

[124] Duane Merrill and Michael Garland. "Merge-based parallel sparse matrix-vector multiplication". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press. 2016, p. 58.

[125] Hans Mittelmann. *Benchmarks for Optimization Software*. Apr. 2020. URL: http://plato.asu.edu/bench.html.

[126] Mahdi Soltan Mohammadi, Kazem Cheshmi, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. "Extending Index-Array Properties for Data Dependence Analysis". In: *Languages and Compilers for Parallel Computing*. Ed. by Mary Hall and Hari Sundar. Cham: Springer International Publishing, 2019, pp. 78–93. ISBN: 978-3-030-34627-0.

[127] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. "Sparse computation data dependence simplification for efficient compiler-generated inspectors". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 594–609.

[128] 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*. 2009, pp. 1–12. DOI: 10.1145/1654059.1654096.

[129] ApS Mosek. *The MOSEK optimization toolbox for MATLAB manual*. 2015.

[130] Ailson P de Moura and Adriano Aron F de Moura. "Newton–Raphson power flow with constant matrices: a comparison with decoupled power flow methods". In: *International Journal of Electrical Power & Energy Systems* 46 (2013), pp. 108–114.

[131] Maxim Naumov. "Parallel incomplete-LU and Cholesky factorization in the preconditioned iterative methods on the GPU". In: *NVIDIA Technical Report NVR-2012-003* (2012).

[132] Maxim Naumov. "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU". In: *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011* 1 (2011).

[133] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[134] Gurobi Optimization. *Inc.,"Gurobi optimizer reference manual," 2015*. 2014.

[135] Abhishek Goud Pandala, Yanran Ding, and Hae-Won Park. "qpSWIFT: A Real-Time Sparse Quadratic Program Solver for Robotic Applications". In: *IEEE Robotics and Automation Letters* 4.4 (2019).

[136] M Papadrakakis and N Bitoulas. "Accuracy and effectiveness of preconditioned conjugate gradient algorithms for large and ill-conditioned problems". In: *Computer methods in applied mechanics and engineering* 109.3-4 (1993), pp. 219–232.

[137] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. "Sparsifying synchronization for high-performance shared-memory sparse triangular solver". In: *International Supercomputing Conference*. Springer. 2014, pp. 124–140.

[138] Roger P Pawlowski, John N Shadid, Joseph P Simonis, and Homer F Walker. "Globalization techniques for Newton–Krylov methods and applications to the fully coupled solution of the Navier–Stokes equations". In: *SIAM review* 48.4 (2006), pp. 700–721.

[139] François Pellegrini and Jean Roman. "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs". In: *International Conference on High-Performance Computing and Networking*. Springer. 1996, pp. 493–498.

[140] Lukas Polok, Viorela Ila, Marek Solony, Pavel Smrz, and Pavel Zemcik. "Incremental Block Cholesky Factorization for Nonlinear Least Squares in Robotics." In: *Robotics: Science and Systems*. 2013.

[141] Alex Pothen and Chunguang Sun. "A mapping algorithm for parallel sparse Cholesky factorization". In: *SIAM Journal on Scientific Computing* 14.5 (1993), pp. 1253–1257.

[142] Alex Pothen and Sivan Toledo. *Elimination Structures in Scientific Computing*. 2004.

[143] Michael James David Powell. "On the quadratic programming algorithm of Goldfarb and Idnani". In: *Mathematical Programming Essays in Honor of George B. Dantzig Part II*. Springer, 1985.

[144] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. "SPIRAL: Code Generation for DSP Transforms". In: *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93.2 (2005), pp. 232–275.

[145] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. "Generation of efficient nested loops from polyhedra". In: *International Journal of Parallel Programming* 28.5 (2000), pp. 469–498.

[146] Michael Rabinovich, Roi Poranne, Daniele Panozzo, and Olga Sorkine-Hornung. "Scalable locally injective mappings". In: *ACM Transactions on Graphics (TOG)* 36.2 (2017), p. 16.

[147] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 519–530.

[148] Lawrence Rauchwerger, Nancy M Amato, and David A Padua. "Run-time methods for parallelizing partially parallel loops". In: *Proceedings of the 9th international conference on Supercomputing*. ACM. 1995, pp. 137–146.

[149] L. Righetti and S. Schaal. "Quadratic programming for inverse dynamics with optimal distribution of contact forces". In: *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*. 2012.

[150] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A Anderson, and Mikhail Smelyanskiy. "Sparso: Context-driven optimizations of sparse linear algebra". In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM. 2016, pp. 247–259.

[151] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jungel, and Siegfried Selberherr. "ViennaCL—linear algebra library for multi-and many-core architectures". In: *SIAM Journal on Scientific Computing* 38.5 (2016), S412–S439.

[152] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[153] Yousef Saad and Andrei V Malevsky. "P-Sparslib: a portable library of distributed memory sparse iterative solvers". In: *Proceedings of Parallel Computing Technologies (PaCT-95), 3-rd international conference, St. Petersburg*. Citeseer. 1995.

[154] Olaf Schenk and Klaus Gärtner. "On fast factorization pivoting methods for sparse symmetric indefinite systems." eng. In: *ETNA. Electronic Transactions on Numerical Analysis [electronic only]* 23 (2006). URL: http://eudml.org/doc/127439.

[155] Olaf Schenk and Klaus Gärtner. "Solving unsymmetric sparse systems of linear equations with PARDISO". In: *Future Generation Computer Systems* 20.3 (2004), pp. 475–487.

[156] Olaf Schenk and Klaus Gärtner. "Two-level dynamic scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems". In: *Parallel Computing* 28.2 (2002), pp. 187–197.

[157] Olaf Schenk, Klaus Gärtner, and Wolfgang Fichtner. "Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors". In: *BIT Numerical Mathematics* 40.1 (2000), pp. 158–176.

[158] Olaf Schenk, Klaus Gärtner, Wolfgang Fichtner, and Andreas Stricker. "PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation". In: *Future Generation Computer Systems* 18.1 (2001), pp. 69–78.

[159] K Schittkowski. "QL: A Fortran code for convex quadratic programming-User guide". In: *Report, Department of Mathematics, University of Bayreuth* (2003).

[160] Michele Segata. *MPC Library*. https://github.com/michele-segata/mpclib. 2019.

[161] Kai Shen, Tao Yang, and Xiangmin Jiao. "S+: Efficient 2D sparse LU factorization on parallel machines". In: *SIAM Journal on Matrix Analysis and Applications* 22.1 (2000), pp. 282–305.

[162] Andrew H Sherman. "Algorithms for sparse Gaussian elimination with partial pivoting". In: *ACM Transactions on Mathematical Software (TOMS)* 4.4 (1978), pp. 330–338.

[163] Jaewook Shin, Mary W Hall, Jacqueline Chame, Chun Chen, Paul F Fischer, and Paul D Hovland. "Speeding up Nek5000 with autotuning and specialization". In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM. 2010, pp. 253–262.

[164] Breannan Smith, Danny M. Kaufman, Etienne Vouga, Rasmus Tamstorf, and Eitan Grinspun. "Reflections on Simultaneous Impact". In: *ACM Trans. Graph.* 31.4 (July 2012). ISSN: 0730-0301.

[165] Intel Software. *OpenMP potential gain definition in intel VTune*. 2018. URL: https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/cpu-metrics-reference/openmp-potential-gain.html (visited on 05/28/2018).

[166] Mahdi Soltan Mohammadi. "Automatic Sparse Computation Parallelization by Utilizing Domain-Specific Knowledge in Data Dependence Analysis". English. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2021-05-13. PhD thesis. 2020, p. 109. ISBN: 9798678143679. URL: http://myaccess.library.utoronto.ca/login?qurl=https%3A%2F%2Fwww.proquest.com%2Fdissertations-theses%2Fautomatic-sparse-computation-parallelization%2Fdocview%2F2451139805%2Fse-2%3Faccountid%3D14771.

[167] Fengguang Song, Asim YarKhan, and Jack Dongarra. "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems". In: *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*. IEEE. 2009, pp. 1–11.

[168] Saeed Soori, Aditya Devarakonda, Zachary Blanco, James Demmel, Mert Gurbuzbalaban, and Maryam Mehri Dehnavi. "Reducing communication in proximal Newton methods for sparse least squares problems". In: *Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

[169] Daniele G Spampinato and Markus Püschel. "A basic linear algebra compiler". In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM. 2014, p. 23.

[170] Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. "OSQP: An operator splitting solver for quadratic programs". In: *Mathematical Programming Computation* (2020), pp. 1–36.

[171] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. "Compile-time composition of run-time data and iteration reorderings". In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 2003, pp. 91–102.

[172] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. "Combining performance aspects of irregular gauss-seidel via sparse tiling". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2002, pp. 90–110.

[173] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. "Sparse tiling for stationary iterative methods". In: *The International Journal of High Performance Computing Applications* 18.1 (2004), pp. 95–113.

[174] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. "The sparse polyhedral framework: Composing compiler-generated inspector-executor code". In: *Proceedings of the IEEE* 106.11 (2018), pp. 1921–1934.

[175] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. "An approach for code generation in the sparse polyhedral framework". In: *Parallel Computing* 53 (2016), pp. 32–57.

[176] Michelle Mills Strout, Fabio Luporini, Christopher D Krieger, Carlo Bertolli, Gheorghe-Teodor Bercea, Catherine Olschanowsky, J Ramanujam, and Paul HJ Kelly. "Generalizing run-time tiling with the loop chain abstraction++". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 1136–1145.

[177] Daniel Sýkora, Ladislav Kavan, Martin Čadík, Ondřej Jamriška, Alec Jacobson, Brian Whited, Maryann Simmons, and Olga Sorkine-Hornung. "Ink-and-Ray: Bas-Relief Meshes for Adding Global Illumination Effects to Hand-Drawn Characters". In: *ACM Trans. Graph.* 33.2 (Apr. 2014). ISSN: 0730-0301. DOI: 10.1145/2591011. URL: https://doi.org/10.1145/2591011.

[178] Tetsuya Takahashi and Christopher Batty. "Monolith: A Monolithic Pressure-Viscosity-Contact Solver for Strong Two-Way Rigid-Rigid Rigid-Fluid Coupling". In: *ACM Trans. Graph.* 39.6 (Nov. 2020). ISSN: 0730-0301. DOI: 10.1145/3414685.3417798. URL: https://doi.org/10.1145/3414685.3417798.

[179] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. "The pochoir stencil compiler". In: *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2011, pp. 117–128.

[180] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. "Collecting performance data with PAPI-C". In: *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[181] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. "A scalable auto-tuning framework for compiler optimization". In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–12.

[182] Ehsan Totoni, Michael T Heath, and Laxmikant V Kale. "Structure-adaptive parallel solution of sparse triangular linear systems". In: *Parallel Computing* 40.9 (2014), pp. 454–470.

[183] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. "XSEDE: accelerating scientific discovery". In: *Computing in science & engineering* 16.5 (2014).

[184] Harmen LA Van Der Spek and Harry AG Wijshoff. "Sublimation: expanding data structures to enable data instance specific optimizations". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2010, pp. 106–120.

[185] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. "Polyhedral code generation in the real world". In: *International Conference on Compiler Construction*. Springer. 2006, pp. 185–201.

[186] Anand Venkat, Mary Hall, and Michelle Strout. "Loop and data transformations for sparse matrix code". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2015, pp. 521–532.

[187] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. "Automating wavefront parallelization for sparse matrix computations". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press. 2016, p. 41.

[188] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. "Non-affine extensions to polyhedral code generation". In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM. 2014, p. 185.

[189] Richard Vuduc, Shoaib Kamil, Jen Hsu, Rajesh Nishtala, James W Demmel, and Katherine A Yelick. "Automatic performance tuning and analysis of sparse triangular solve". In: ICS. 2002.

[190] Andreas Wächter and Lorenz T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Mathematical Programming* 106.1 (Mar. 2006). ISSN: 1436-4646.

[191] Richard A Waltz and Jorge Nocedal. "KNITRO 2.0 User's Manual". In: *Ziena Optimization, Inc.[en ligne] disponible sur http://www. ziena. com* 7 (2004).

[192] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. "Intel math kernel library". In: *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[193] Xinliang Wang, Weifeng Liu, Wei Xue, and Li Wu. "SwSpTRSV: A Fast Sparse Triangular Solve with Sparse Level Tile Layout on Sunway Architectures". In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '18. Vienna, Austria: Association for Computing Machinery, 2018, pp. 338–353. ISBN: 9781450349826. DOI: 10.1145/3178487.3178513. URL: https://doi.org/10.1145/3178487.3178513.

[194] Nicholas J. Weidner, Kyle Piddington, David I. W. Levin, and Shinjiro Sueda. "Eulerian-on-Lagrangian Cloth Simulation". In: *ACM Trans. Graph.* 37.4 (July 2018). ISSN: 0730-0301.

[195] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. "Optimization of sparse matrix–vector multiplication on emerging multicore platforms". In: *Parallel Computing* 35.3 (2009), pp. 178–194.

[196] Elizabeth Wong. *Active-set methods for quadratic programming*. University of California, San Diego, 2011.

[197] Stephen J Wright. *Primal-dual interior-point methods*. Vol. 54. Siam, 1997.

[198] Z Xianyi. *OpenBLAS: an optimized BLAS library*. 2016.

[199] Jiaxian Yao, Danny M. Kaufman, Yotam Gingold, and Maneesh Agrawala. "Interactive Design and Stability Analysis of Decorative Joinery for Furniture". In: *ACM Trans. Graph.* 36.4 (Mar. 2017). ISSN: 0730-0301.

[200] Yu-Hong Yeung, Jessica Crouch, and Alex Pothen. "Interactively Cutting and Constraining Vertices in Meshes Using Augmented Matrices". In: *ACM Trans. Graph.* 35.2 (Feb. 2016). ISSN: 0730-0301.

[201] Changxi Zheng and Doug L. James. "Toward High-Quality Modal Contact Sound". In: *ACM Trans. Graph.* SIGGRAPH '11 (July 2011).

[202] Yufeng Zhu, Robert Bridson, and Danny M. Kaufman. "Blended Cured Quasi-Newton for Distortion Optimization". In: *ACM Trans. Graph.* 37.4 (July 2018). ISSN: 0730-0301.

[203] Sicong Zhuang and Marc Casas. "Iteration-fusing conjugate gradient". In: *Proceedings of the International Conference on Supercomputing*. 2017, pp. 1–10.

[204] Xiaotong Zhuang, Alexandre E Eichenberger, Yangchun Luo, Kevin O'Brien, and Kathryn O'Brien. "Exploiting parallelism with dependence-aware scheduling". In: *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE. 2009, pp. 193–202.

[205] Intel Developer Zone. "Intel VTune Amplifier, 2017". In: *Documentation at the URL: https://software.intel.com/en-us/intel-vtune-amplifier-xe-support/documentation* ().